Typové systémy pro netypované jazyky

Libor Škarvada

Obsah

1 Úvod

Text obsahuje přehled základních výsledků o typových systémech, rozdělení nejdůležitějších typových kalkulů a jejich různých rozšíření a modifikací, které se jednak objevily v literatuře za posledních cca 30 let, jednak se staly základem typových systémů existujících programovacích jazyků. Kde je to možné, upozorňujeme na vztahy a souvislosti mezi některými kalkuly.

Dále se věnujeme vlastnostem těchto kalkulů, s důrazem na vlastnosti a jevy souvisícími s jazyky, na jejichž typové systémy se bude další práce zaměřovat. Jsou to zejména parametrický polymorfismus, podtypy a kombinace s tzv. dynamickými typy.

Další oblastí je typová kontrola a typové odvození v těchto kalkulech a jejich algoritmická rozhodnutelnost.

Jelikož se v praktickém programování používají a jsou oblíbené netypované jazyky, všímáme si možností, jak rozšířit jejich triviání typový systém doplněním o netriviální typy. Důvodem pro tato rozšíření je snaha učinit tyto jazyky typově bezpečnějšími, a tím zkrátit a zefektivnit vývojový cyklus. Jednou ze zajímavých a prakticky schůdných možností jsou tzv. *částečné typové systémy*, které kombinují statické typy s dynamickými kontrolami běhových příznaků. Ty jsou využity v několika praktických nástrojích, z nichž tři nejznámější také popíšeme.

Nastíníme zamýšlené směry zkoumání typových kalkulů, zejména se zaměřením na kalkuly rozšiřující triviální typový systém netypovaných jazyků.

2 Typové systémy

Typový systém je dle obecné neformální definice formalismus popisující konstrukci typů, vztahy mezi typy, operace na typech a pravidla pro přiřazení typů výrazům. Přesné definice jsou součástí konkrétních typových systémů (příklady některých jsou uvedeny v odst. 2.3).

V netypovaných programovacích jazycích se typy hodnot nerozlišují. Lze však na ně nahlížet jako na typované jazyky s triviálním typovým systémem. Triviální systém obsahuje jediný typ, který je společný všem datům. Výstižnější název pro netypované jazyky je proto *jednotypové*[1] jazyky.

[1] *unityped* vs. *untyped*

2.1 Odvozovací pravidla

Formálně je typový systém definován pravidly, která se zapisují podobně jako odvozovací pravidla v logických systémech:

$$\frac{a_1 \quad \ldots \quad a_k}{a_0}$$

Přitom $a_1, \ldots, a_k$ jsou *předpoklady* a a_0 je *závěr* pravidla. Předpokladům a závěru se souhrnně říká *aserce*. Například aserce tvaru $\Gamma \vdash \sigma : \star$ vyjadřuje skutečnost, že nějaký typový výraz σ je dobře utvořeným typem, aserce $\Gamma \vdash v : \sigma$ říká, že hodnota v má typ σ, anebo aserce $\Gamma \vdash \sigma \equiv \tau$ říká, že mezi dvěma typy platí vztah $\equiv$. Pokud aserce závisí na volných proměnných v ní obsažených, vymezují se typy těchto proměnných tzv. *kontextem*. Kontext $\Gamma = \{(x_1, t_1), \ldots, (x_n, t_n)\}$ je konečná množina uspořádaných dvojic, v níž $x_1, \ldots, x_n$ jsou navzájem různé proměnné a $t_1, \ldots, t_n$ jejich typy. V aserci se kontext zapisuje bez závorek a se znakem : mezi proměnnou a typem: $x_1 : t_1, \ldots, x_n : t_n$. Prázdný kontext ze zápisu aserce vynecháváme.

Podle bohatosti typového systému existuje více druhů pravidel, zejména to jsou následující:

- Pravidla pro konstrukci typů, kde aserce vyjadřují skutečnost, že nějaký typový výraz je dobře utvořeným typem. Obecněji, v systémech s více sortami, aserce říká, že výraz dané sorty je dobře utvořen.
- Pravidla, podle nichž se přiřazují typy (hodnotovým) výrazům. Aserce v tomto případě vyjadřují tvrzení, že daný výraz je daného typu. V systémech s více sortami se těmito asercemi vyjadřuje, že výraz označující nějakou sortu patří do jiné, vyšší sorty (například $\Gamma \vdash \sigma : \star \to \star$ říká, že typ σ patří do druhu $\star \to \star$).
- Pravidla zavádějící relace mezi typy. Určují, kdy jsou například dva typy ekvivalentní, nebo, v typových systémech s podtypy, kdy je nějaký typ podtypem jiného typu.

2.2 Hierarchie sort

Jednoduché typové systémy rozeznávají jen dvě *sorty*: hodnoty a typy. Pokročilé typové systémy mohou mít celou hierarchii sort — rozeznávají typy hodnot, typy typů (neboli tzv. druhy), typy druhů atd.

Hodnoty jsou sémantické entity — prvky nějakého universa [16, kap. 1.1]. Jejich syntaktickým protějškem jsou výrazy.

Typy jsou entity definované typovým systémem jazyka a slouží pro klasifikaci hodnot: v jazycích s typy má každá hodnota nějaký typ. Syntakticky se typy popisují typovými výrazy, jejichž gramatika je dána typovým systémem.

V typových systémech s netriviálnílní gramatikou typových výrazů je potřeba klasifikovat také typy a zavádí se třetí úroveň, tzv. *druhy*.

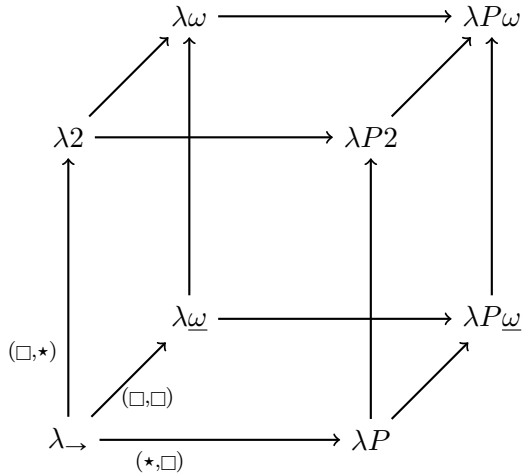
Hierarchie sort v praktických programovacích jazycích jako Haskell [4] nebo ML [19] zpravidla končí u druhů. S rostoucí úrovní sorty se totiž zjednodušuje gramatika výrazů, které tuto sortu popisují, a od jisté úrovně je triviální. Například v Haskellu jazyk sort obsahuje jedinou sortu $\Box$ všech druhů. Avšak v jazycích Agda [20], Idris [11], Coq [12] pokračuje hierarchie sort do nekonečna.

V systémech s nekonečně mnoha sortami mají výrazy všech sort stejnou syntax. Naopak Haskell nebo ML mají různou syntax pro hodnotové a typové výrazy. V těchto jazycích jsou totiž syntaktické kategorie pro hodnotové, typové i druhové výrazy navzájem disjunktní: hodnota nesmí být součástí typového výrazu apod.

2.3 Barendregtova lambda-krychle

Barendregtova lambda-krychle [8, 9] je elegantní klasifikací typových systémů funkcionálních kalkulů se dvěma sortami — hodnotami a typy. Vychází z jednoduše typovaného lambda kalkulu, jenž tvoří její počáteční vrchol a jehož typový systém krychle zobecňuje ve třech dimenzích:

- $(\Box, \star)$ závislost hodnot na typech, vedoucí k parametrickému polymorfismu
- $(\Box, \Box)$ závislost typů na typech, systémy s typovými funkcemi
- $(\star, \Box)$ závislost typů na hodnotách



Vrcholy krychle pak odpovídají kalkulům:

$\lambda_{\rightarrow}$	jednoduše typovaný lambda kalkul
$\lambda 2$	polymorfní lambda kalkul, hodnoty parametrizované typy
$\lambda\underline{\omega}$	kalkul s obecnými typovými funkcemi
λP	kalkul s hodnotově závislými typy
$\lambda\omega$	polymorfní kalkul s obecnými typovými funkcemi
$\lambda P\underline{\omega}$	kalkul se závislými typy a obecnými typovými funkcemi
$\lambda P2$	polymorfní kalkul se závislými typy
$\lambda P\omega$	kalkul konstrukcí

2.4 Typová kontrola, otypování, neprázdnost typu

Otázky souvisící se vztahem mezi hodnotou v a typem σ jsou

$v : \sigma$	typová kontrola (*Má hodnota v deklarovaný typ σ?*)
$v : ?$	odvození typu neboli otypování (*Jaký typ má hodnota v?*)
$? : \sigma$	neprázdnost neboli obydlenost typu (*Je typ σ neprázdný?*)

Podle složitosti kalkulu může či nemusí existovat algoritmus, který na uvedené otázky odpovídá. Například kalkuly v lambda-krychli mají rozhodnutelnou typovou kontrolu i otypování, ale, jak uvidíme v odst. 2.5, po malé modifikaci syntaxe — vymazání typů z termů — se stávají otypování i typová kontrola nerozhodnutelnými pro většinu vrcholů krychle.

2.5 Churchova a Curryho varianta typovaných kalkulů

Kalkuly v lambda-krychli jsou tzv. *kalkuly s explicitními typy*, nazývané též kalkuly podle Churche. V nich jsou totiž typy hodnotových proměnných, druhy typových proměnných atd. syntaktickou částí termu. V praktických jazycích těmto kalkulům odpovídá povinnost psát explicitní typovou anotaci pro každou proměnnou použitou v programu. Algoritmus otypování je triviální, protože termy si svoje typy nesou s sebou.

Alternativní variantou jsou *kalkuly s implicitními typy* nazývané též kalkuly podle Curryho.[2] V jejich termech se zápisy typů nevyskytují. Termy se syntakticky tvoří stejně jako v netypovaném lambda kalkulu.

Rozdíl je vidět například na zápisu polymorfní identické funkce $\lambda x.x$ a na její aplikaci na nulu: $(\lambda x.x)\,0$. Syntax je stejná jako u netypovaného lambda kalkulu, veškeré informace o typech podtermů je nutno odvodit.

[2] calculi à la Church vs. calculi à la Curry

Naproti tomu Churchova varianta stejného kalkulu má zvláštní termy pro typovou abstrakci a pro typovou specializaci, viz [10, app. A]. Polymorfní identická funkce je vyjádřena termem $\Lambda\alpha{:}\star\lambda x{:}\alpha.\, x$, její aplikace na nulu ohsahuje explicitní typovou specializaci na číselný typ: $(\Lambda\alpha{:}\star\lambda x{:}\alpha.\, x)\ \mathit{Nat}\ 0$.

Typované funkcionální jazyky používané v praxi jsou Curryho variantou kalkulů. Typové anotace v nich není nutno uvádět. Algoritmus otypování je zde těžší než u kalkulů s explicitními typy a pro složitější typové systémy ani neexistuje.

Například pro polymorfní lambda kalkul $\lambda 2$ (kalkul s explicitními typy) je rozhodnutelnost typové kontroly stejná jako rozhodnutelnost typového odvození, ale pro Curryho variantu stejného kalkulu jsou tyto problémy nerozhodnutelné — podrobněji viz odst. 2.7.

2.6 Směry zobecnění a rozšíření typových systémů

2.6.1 Vyšší sorty, PTS

Přirozeným rozšířením typových systémů z lambda-krychle je přidání dalších sort. Takzvané *ryzí typové systémy*[3] pracují s nekonečně mnoha sortami.

Barendregt v [9] definuje ryzí typový systém pomocí malé množiny schémat odvozovacích pravidel. Tato schémata jsou parametrizována binární relací na sortách. Lambda-krychle je speciálním případem: uvažuje pouze dvě sorty, $\star$ (druh typů) a $\Box$ (sortu druhů), a na nich všechny binární relace, které obsahují dvojici $(\star, \star)$. Těch je osm a odpovídají výše uvedeným kalkulům. Přidávání dalších sort do krychle vede k exponenciální expanzi a rychle znepřehlední situaci. Kalkuly s pravidly kombinujícími více než dvě sorty nebyly studovány.

2.6.2 Induktivně a koinduktivně definované typy

Konkrétní jazyk může obsahovat například pravidla

$$\mathit{Nat} : \star \qquad\qquad \mathit{0} : \mathit{Nat} \qquad\qquad \frac{x : \mathit{Nat}}{S\ x : \mathit{Nat}}$$

Ta definují *Nat* jako množinově nejmenší typ obsahující předepsané termy. Avšak jazyk může též obsahovat obecnou konstrukci, pomocí níž lze taková pravidla libovolně přidávat. Potom mluvíme o jazyce s *induktivními typy*.

[3] pure type systems, PTS

V jazyce Coq jsou výše uvedená pravidla zapsána definicí typu *Nat*

$$\begin{aligned}&\textbf{Inductive } \mathit{Nat} : \mathsf{Type} \ :=\\ &\quad |\ \ \mathit{0} \ : \ \mathit{Nat}\\ &\quad |\ \ \mathit{S} \ : \ \mathit{Nat} \to \mathit{Nat}\end{aligned}$$

jejímž významem je nejmenší pevný bod $\boldsymbol{\mu}F$ typové funkce $F\,a = \mathit{Unit} + (\mathit{Unit} \times a)$, který je isomorfní s typem přirozených čísel a jehož uzavřené termy jsou tzv. *číslovky* $\mathit{0},\ \mathit{S}\,\mathit{0},\ \mathit{S}(\mathit{S}\,\mathit{0}), \ldots$

Duální konstrukce definuje nově přidaný typ jako množinově největší typ obsahující předepsané termy. Tato konstrukce zavádí *koinduktivní typy.*

Modifikujeme-li předchozí příklad v Coqu,

$$\begin{aligned}&\textbf{CoInductive } \mathit{Nat}' \ : \mathsf{Type} \ :=\\ &\quad |\ \ \mathit{0} \ : \ \mathit{Nat}'\\ &\quad |\ \ \mathit{S} \ : \ \mathit{Nat}' \to \mathit{Nat}'\end{aligned}$$

získáme typ, který je *největším* pevným bodem $\boldsymbol{\nu}F$ stejné typové funkce a je isomorfní s typem přirozených čísel doplněným o term $S(S(\ldots\ \ldots)) = (\textbf{let } \mathit{inf}{=}S\,\mathit{inf} \textbf{ in } \mathit{inf})$, neboli s typem přirozených čísel rozšířených o nekonečný prvek.

Kalkul, který vznikne přidáním induktivních a koinduktivních typů do kalkulu konstrukcí ($\lambda P\omega$), se nazývá *kalkul induktivních konstrukcí* a je základem jazyka Coq [13].

Z kombinace koinduktivních typů s podtypy (viz odst. 2.6.3) vycházejí typové systémy pro objektově orientované programovací jazyky, zejména varianty tzv. *sigma kalkulu* [7].

2.6.3 Podtypy a subsumpce

Typové systémy s podtypy jsou základem typovaných objektově orientovaných jazyků. Dosud zmíněné typové systémy uvažovaly nanejvýš jednu relaci mezi typy jedné sorty, a to rovnost typů. Přidáním reflexivní a transitivní relace *podtypu* $<\!:$ (resp. její inverze, relace *nadtypu*, $:\!>$) zavádíme kvaziuspořádání na typech.

$$A <\!: A \qquad\qquad \frac{A <\!: B \qquad B <\!: C}{A <\!: C}$$

Kalkul často obsahuje speciální typy $\bot$ a $\top$, které jsou nejmenším, resp. největším pvrkem relace podtypu:

$$\bot <\!: A \qquad\qquad A <\!: \top$$

Důležitým otypovávacím pravidlem v systému s podtypy je *pravidlo subsumpce*: má-li hodnota a typ A, pak jsou jejím typem i všechny nadtypy typu A.

$$\frac{a : A \qquad A <: A'}{a : A'}$$

Pravidlo a jeho důsledky se nazývají *podtypovým polymorfismem*.

Typové operace + a × mají následující pravidla podtypování, která vyjadřují monotonii součtu a součinu vzhledem k relaci $<:$, neboli tzv. *kovarianci* těchto typových operací v obou argumentech.

$$\frac{A <: A' \qquad B <: B'}{A \times B <: A' \times B'} \qquad\qquad \frac{A <: A' \qquad B <: B'}{A + B <: A' + B'}$$

Typová šipka $\to$ je také binární typová operace. Narozdíl od součtu a součinu je však ve svém prvním (levém) typovém argumentu *kontravariantní*, tj. antimonotonní vzhledem k relaci $<:$. Ve druhém typovém argumentu je kovariantní. Pravidlo podtypování typů funkcí je tedy

$$\frac{A <: A' \qquad B <: B'}{A' \to B <: A \to B'}$$

Obecněji, jsou-li $F : \star \to \star$, $G : \star \to \star$ typové funkce, pak pravidla

$$\frac{A <: A'}{F\ A <: F\ A'} \qquad\qquad \frac{A <: A'}{G\ A' <: G\ A}$$

vyjadřují *kovarianci* funkce F a *kontravarianci* funkce G.

Pro součet a pro součin jsou důležitá pravidla extenze

$$\frac{A <: A'}{A <: A' + B} \qquad \frac{B <: B'}{B <: A + B'} \qquad \frac{A <: A'}{A \times B <: A'} \qquad \frac{B <: B'}{A \times B <: B'}$$

Poslední dvě pravidla říkají v jazycích s tzv. *objekty* nebo *záznamy* lze do objektu (záznamu) typu A přidat novou položku a získat tím hodnotu podtypu typu A.

2.6.4 Průnikové typy a typy sjednocení

Relace podtypu je obecně kvaziuspořádání. Pokud tvoří dokonce svaz, můžeme přidat svazové operace infima a suprema, což jsou, v souladu s množinovou interpretací

typů jakožto množin hodnot, průnikové typy $A \wedge B$ a typy sjednocení $A \vee B$. Typy se chovají v souladu se svou množinovou interpretací:

$$A \wedge B <: A \qquad A \wedge B <: B \qquad \frac{C <: A \qquad C <: B}{C <: A \wedge B}$$

$$A <: A \vee B \qquad B <: A \vee B \qquad \frac{A <: C \qquad B <: C}{A \vee B <: C}$$

Z těchto pravidel vyplývají rovnosti pro operace s největším a nejmenším typem:

$$\begin{array}{lllll} A \wedge \bot & = & \bot \wedge A & = & \bot \\ A \vee \bot & = & \bot \vee A & = & A \end{array} \qquad \begin{array}{lllll} A \wedge \top & = & \top \wedge A & = & A \\ A \vee \top & = & \top \vee A & = & \top \end{array}$$

Z vlastností svazových operací plyne, že pro kovariantní typovou funkci $F : \star \to \star$ platí

$$\begin{array}{lll} F\,(A \wedge B) & <: & F\,A \;\wedge\; F\,B \\ F\,(A \vee B) & :> & F\,A \;\vee\; F\,B \end{array}$$

Podobně pro kontravariantní typovou funkci $G : \star \to \star$ máme

$$\begin{array}{lll} G\,(A \wedge B) & :> & G\,A \;\vee\; G\,B \\ G\,(A \vee B) & <: & G\,A \;\wedge\; G\,B \end{array}$$

Speciálně pro typovou šipku $\to$ po dvojím použitím těchto nerovností máme

$$\begin{array}{lll} (A \to B) \;\wedge\; (C \to D) & :> & (A \vee C) \;\to (B \wedge D) \\ (A \to B) \;\vee\; (C \to D) & <: & (A \wedge C) \;\to (B \vee D) \end{array}$$

Průnikové typy a typy sjednocení se uplatňují v typových systémech jazyků pracujících se záznamy nebo objekty. Záznam je uspořádaná n-tice indexovaná konečnou množinou *selektorů* $\{a_1, \ldots, a_n\}$. Záznam se složkami $v_1, \ldots, v_n$ zapisujeme $\{a_1{=}v_1, \ldots, a_n{=}v_n\}$ a jeho typ $\{a_1 : \sigma_1, \ldots, a_n : \sigma_n\}$.

Nechť $1 \leqslant k \leqslant m \leqslant n$ a mějme dva záznamové typy $A = \{a_1 : \sigma_1, \ldots, a_m : \sigma_m\}$ a $B = \{a_{k+1} : \tau_{k+1}, \ldots, a_n : \tau_n\}$, jejichž společné selektory jsou $a_{k+1}, \ldots, a_m$. Průnik těchto záznamových typů definujeme:

$$\begin{array}{lll} A \;\wedge\; B & = & \{\; a_1 : \sigma_1,\; \ldots,\; a_k : \sigma_k, \\ & & \quad a_{k+1} : \sigma_{k+1} \wedge \tau_{k+1},\; \ldots,\; a_m : \sigma_m \wedge \tau_m, \\ & & \quad a_{m+1} : \tau_{m+1},\; \ldots,\; a_n : \tau_n \qquad\qquad \} \end{array}$$

Definice sjednocení záznamových typů je duální — typ sjednocení obsahuje pouze společné selektory:

$$A \ \vee \ B \ \ = \ \ \{ \ a_{k+1} : \sigma_{k+1} \vee \tau_{k+1}, \ \ldots, \ a_m : \sigma_m \vee \tau_m \ \}$$

Příklad: Nechť typy selektorů jsou $n : \mathit{Int}$, $c : \mathit{Char}$, $s : \mathit{String}$, typ logické konjunkce je $\texttt{\&\&} : \mathit{Bool} \times \mathit{Bool} \rightarrow \mathit{Bool}$ a funkce g je použita ve výrazu

$$g\,(\{n{=}3, s{=}\texttt{"abc"}\}) \ \ \texttt{\&\&} \ \ g\,(\{n{=}6, c{=}\texttt{'z'}\})$$

Potom odvozené typy funkce g v obou aplikacích jsou $\{n : \mathit{Int}, s : \mathit{String}\} \rightarrow \mathit{Bool}$, resp. $\{n : \mathit{Int}, c : \mathit{Char}\} \rightarrow \mathit{Bool}$. Typ funkce g bude průnikem obou typů:

$$\begin{aligned} g \ \ : \ \ & (\{n : \mathit{Int}, s : \mathit{String}\} \rightarrow \mathit{Bool}) \ \wedge \ (\{n : \mathit{Int}, c : \mathit{Char}\} \rightarrow \mathit{Bool}) \ \ \Rightarrow \\ & \{n : \mathit{Int}, s : \mathit{String}\} \vee \{n : \mathit{Int}, c : \mathit{Char}\} \ \rightarrow \ \mathit{Bool} \ \ = \\ & \{n : \mathit{Int}\} \ \rightarrow \ \mathit{Bool} \end{aligned}$$

2.7 Specializace typových systémů

Kromě zobecňování lze typové systémy zužovat a omezovat jejich vyjadřovací schopnost. To je vhodné, když například pro plnou verzi kalkulu algoritmus otypování neexistuje, není znám, nebo je neefektivní, ale pro restringovanou verzi použitelný algoritmus existuje. Omezení typového systému neznamená nutně omezení vyjadřovací síly vlastního jazyka (jazyka hodnotových termů).

Typickým příkladem je Hindleyho-Milnerův typový systém pro Curryho variantu polymorfního lambda kalkulu [14, 18]. Nazývá se též typovým systémem s *mělkým* polymorfismem nebo s *polymorfismem ranku 1*, protože typové proměnné v typových termech smějí být kvantifikovány jen na nejvyšší úrovni.

V plné obecně polymorfní verzi kalkulu jsou typy definovány následujícími pravidly (typ ι je bázový a typy σ jsou obecné).

$$\iota : \star \qquad \frac{\sigma_1 : \star \qquad \sigma_2 : \star}{\sigma_1 \rightarrow \sigma_2 : \star} \qquad \frac{\sigma_1 : \star \qquad \alpha{:}\sigma_1 \ \vdash \ \sigma_2 : \star}{\forall \alpha.\sigma_2 : \star}$$

V Hindleyho-Milnerově systému je definice typů stratifikovaná — rozdělená na nekvantifikované typy τ pomocného druhu TS a na obecné typy σ druhu $\star$.

$$\iota : \mathsf{TS} \qquad \frac{\tau_1 : \mathsf{TS} \qquad \tau_2 : \mathsf{TS}}{\tau_1 \rightarrow \tau_2 : \mathsf{TS}} \qquad \frac{\tau : \mathsf{TS}}{\tau : \star} \qquad \frac{\sigma_1 : \star \qquad \alpha{:}\sigma_1 \ \vdash \ \sigma_2 : \star}{\forall \alpha.\sigma_2 : \star}$$

Přínosem Hindleyho-Milnerovy restrikce typového systému je existence efektivního algoritmu pro nalezení (z hlediska parametrického polymorfismu) nejobecnějšího typu. Pro Curryho variantu polymorfního lambda kalkulu s obecným polymorfismem neexistuje algoritmus pro otypování ani algoritmus pro typovou kontrolu.

Pro jednotlivé varianty kalkulu $\lambda 2$ a pro obecný a mělký polymorfismus je rozhodnutelnost problému otypování ekvivalentní rozhodnutelnosti problému typové kontroly. Jednotlivé varianty shrnuje následující tabulka.

typová kontrola, otypování	Church	Curry
mělký polymorfismus	*rozhodnutelné*	*rozhodnutelné*
obecný polymorfismus	*rozhodnutelné*	*nerozhodnutelné*

Důkaz rozhodnutelnosti pro Churchův kalkul je snadný, protože informace pro otypování je syntakticky obsažena v termech. Konstruktivní důkaz rozhodnutelnosti pro Hindleyho-Milnerův polymorfismus v Curryho kalkulu podal R. Milner v [18]. Nerozhodnutelnost v obecně polymorfním Curryho kalkulu dokázal J. Wells v [25].

Hindleyho-Milnerův mělký polymorfismus je základem typového systému řady funkcionálních programovacích jazyků včetně ML a Haskellu.

2.8 Algoritmy typového odvození

2.8.1 Typové odvození v kalkulu s mělkým polymorfismem

Hindleyho-Milnerův typový systém [18] byl uveden v odst. 2.7.

Povšimněme si, že omezení na mělký polymorfismus je velmi restriktivní, protože (bez dodatečných rozšíření) nelze polymorfismu využít vůbec. Nechť $A : \star$, $B : \star$ jsou dva typy, $a : A$, $b : B$ jsou dvě hodnoty těchto typů, $id : \forall\alpha.\alpha \to \alpha$ je polymorfní identická funkce, jejíhož polymorfismu chceme využít ve výrazu, kde je aplikovaná v kontextu různých typů. Výraz $(id\, a,\ id\, b)$ nečiní problémy a je v Curryho variantě polymorfního lambda kalkulu snadno otypovatelný typem $A \times B$.

Ale ekvivalentní výraz $(\lambda g.(g\, a,\ g\, b))\ id$ otypovatelný není: abychom vyjádřili skutečnost, že proměnná g v lokální definici je polymorfní, musela by lambda abstrakce $\lambda g.(g\, a,\ g\, b)$ mít typ $(\forall\alpha.\alpha \to \alpha) \to A \times B$, což není mělký Hindleyho-Milnerův typ.

V Hindleyho-Milnerově kalkulu se tato potíž řeší zavedením termu **let** $x = u$ **in** e, který popisuje lokální definici v termu e a je sémanticky ekvivalentní aplikaci $(\lambda x.e)\, u$. Otypovávací pravidlo pro nový term je

$$\frac{u : \forall\alpha.\sigma_1 \qquad x{:}\forall\alpha.\sigma_1 \vdash e : \sigma_2}{\textbf{let } x = u \textbf{ in } e : \sigma_2}$$

Pak neotypovatelný term $(\lambda g.(g\ a,\ g\ b))\ id$ nahradíme termem **let** $g = id$ **in** $(g\ a,\ g\ b)$, který už je otypovatelný mělkým typem $A \times B$.

Term $(\lambda g.(g\ a,\ g\ b))\ id$ je totiž kombinací aplikace a abstrakce a problematický typ se vyskytne při odvozování právě mezi použitím obou pravidel. Spojením obou částí do jednoho termu **let** se problém s nepříjemným „mezitypem“ obejde.

Klasický Hindleyho-Milnerův algoritmus typového odvození byl v průběhu let různě modifikován a doplňován ve snaze přizpůsobit ho typovým systémům praktických jazyků.

2.8.2 Typové odvození s typovými třídami

Typové třídy jsou význačným rysem typového systému Haskellu a jsou v jazyce přítomny od začátku jeho existence, tj. od počátku 90. let. Ostatní funkcionální jazyky (zejména ML) je od Haskellu přebraly. Pionýrský článek [15], v němž je S. Peyton Jones definoval a popsal jejich vlastnosti i význam, byl publikován až několik let po vzniku Haskellu.

Typové třídy slouží k omezení polymorfismu. Univerzální kvantifikaci $\forall\alpha.\sigma$ zužují na

$$\forall\alpha.(\mathsf{C}\,\alpha \Rightarrow \sigma)$$

kde predikát $\mathsf{C}\,\alpha$ ohraničuje prostor typů, jež lze dosazovat za typovou proměnnou α.

Díky závislosti na hodnotách — definicích metod v různých instancích, lze typovými třídami formálně přesně popsat přetížení, s nímž se před zavedením typových tříd zacházelo pouze nesystematicky.[4] Například v Haskellu definice typové třídy Monoid a dvou z jejích mnoha instancí jsou

class Monoid a	**instance** Monoid *Integer*	**instance** Monoid *String*
where	**where**	**where**
$\varepsilon : a$	$\varepsilon = 0$	$\varepsilon =$ ""
$(\diamond) : a \to a \to a$	$(\diamond) = (+)$	$(\diamond) = (+\!\!+)$

Deklarace instancí systematicky popisují přetížení symbolů ε a $\diamond$, v prvním případě nulou a sčítáním a ve druhém případě prázdným řetězcem a řetězcovým spojováním.

V dalších zobecněních [17] typová třída může mít libovolnou aritu, doménovými druhy jejích parametrů nemusí být $\star$, ale libovolné vyšší druhy, dále může záviset nejen na hodnotách (metodách), ale i na typech apod.

[4] Odtud také někdy užívaný název *ad-hoc polymorphism* pro přetížení

2.8.3 Odvozování obecně polymorfních typů

Obecný polymorfismus se též nazývá *polymorfismus obecného ranku*, kde rankem se rozumí úroveň zanoření univerzálního kvantifikátoru u typové proměnné. Už jsme zmínili, že otypování v obecně polymorfním systému lambda kalkulu s implicitním typováním rozhodnutelné není. Nicméně například v extenzích Haskellu, které zavádí progresívní kompilátor GHC [1], se polymorfismus obecného ranku používá, protože se brzy zjistilo, že je užitečný.

Klasickým příkladem je typ pro imperativní akce s lokálním stavem a typ funkce, která tyto akce provádí. Akce typu $ST\ s\ a$ má lokální přístup ke stavu typu s a po provedení vrací výslednou hodnotu typu a. Provedení akce zajistí funkce $runST$, jejíž typ není mělký, ale má rank 2:

$$runST\ :\ \forall a.(\forall s.\, ST\ s\ a) \rightarrow a$$

Důvodem takového otypování je potřeba učinit typ stavu s ve funkci nezávislým na okolí, zejména na typu a. Funkce $runST$ je polymorfní jen v typu a; to znamená, že aplikace na danou akci může za a dosadit konkrétní typ, ale dosazovat typ stavu za proměnnou s při aplikaci nelze.

Otypovávací algoritmus Haskellu s extenzí na polymorfismus obecného ranku teoreticky může selhat, ale v praxi to nečiní problémy, protože při neúspěchu se pouze vyžádá explicitní „ruční“ otypování funkce, na níž algoritmus neuspěl. V uvedeném příkladě je to typová anotace funkce $runST$, s jejíž pomocí algoritmus může typy odvodit.

3 Dynamické jazyky

Mnohé v praxi používané programovací jazyky jsou netypované. Určitou kontrolu konsistence zpracovávaných dat však provádějí, a to za běhu programu. Proto se tyto jazyky též nazývají *dynamické*.[5] Součástí dat je informace o typu, tzv. běhový příznak, který operace testují a větví podle něj výpočet, například do různých instancí přetížené funkce.

Poznámka: Dynamické jazyky mají svou terminologii, v níž se samy označují za typované. Pod pojmem „typ“ však rozumějí právě běhový příznak. Naopak v teorii typů pojmem typy rozumíme pouze *statické* typy, které jsou při kompilaci vymazány a v době běhu se v hodnotách neobjeví. Otypovaná funkce je vždy aplikována jen na přípustné argumenty a při běhu se žádné příznaky netestují.

Netypované jazyky mají pro praktické programování některé nevýhody:

[5] Jiné označení pro používání běhových příznaků je *duck typing*.

- absence typů komplikuje nebo znemožňuje potenciální transformace kódu, zejména optimalizace,
- běhové kontroly zpomalují výpočet,
- delší a dražší vývoj programů — chyby se odhalují pozdě

Přesto jsou netypované jazyky oblíbené a rozšířené. První dvě nevýhody lze zmírnit použitím výkonnějších strojů, třetí lze zmenšit důkladným testováním a dostatečně širokou sadou testovacích dat. Nicméně jde jen o zmírnění a nikoli o eliminaci nevýhod.

Odkládání kontrol do doby běhu je popřením zásady, že chyby je třeba najít a odstranit v co nejranější fázi procesu vývoje programu. Proto pro některé používané netypované jazyky vznikly extenze, které do jazyka typy přidávají. Jsou popsány v sekci 4.

3.1 Triviální typování

Název *netypované* jazyky není zcela přesný, protože tyto jazyky lze považovat za jazyky s triviálním typovým systémem, majícím jediný typ a pravidlo říkající, že všechny hodnoty jsou tohoto univerzálního typu. (Viz poznámku v sekci 2.)

$$\top : \star \qquad\qquad e : \top$$

Z praktického hlediska triviální typový systém není nijak užitečný. Má však teoretický význam, například při klasifikaci a porovnávání typových systémů.

3.2 Částečné typování

Existující extenze dynamických jazyků (Flow, Hack, MyPy, viz sekci 4) neusilují o úplné vyloučení běhových kontrol a o plně statické typování. Jejich otypovávací algoritmus je částečný a pro výrazy, na kterých selže, generuje běhové kontroly. Tyto systémy tedy kombinují statické typy s běhovými příznaky. Nazývají se *systémy s částečným typováním.*[6] Dva různé takové systémy popsali Thatte v [24] a Siek a Taha v [23].

Částečného typování lze dosáhnout doplněním běhových příznaků do (staticky) typovaného jazyka. anebo naopak, přidáním statických typů do netypovaného (dynamického) jazyka. Druhý směr je zpravidla obtížnější.

Při přidávání běhových příznaků do jazyka s dostatečně bohatým typovým systémem vystačíme s prostředky jazyka a tuto modifikaci lze realizovat například formou knihovny. K tomu je potřeba:

[6] doslova *s postupným typováním*, *gradual typing*

- vhodně reprezentovat typy σ hodnotami $repr(\sigma)$,
- hodnoty $v : \sigma$ nahradit uspořádanými dvojicemi $(v, repr(\sigma))$,
- definovat konverzní funkce, pro převod mezi hodnotami bez příznaku a s příznakem,
- definovat funkci pro aplikaci funkce na argument s příznakem, která zabezpečí běhovou kontrolu.

Například v Haskellu toto přidání částečného typování řeší knihovna Data.Dynamic. Je však třeba zdůraznit, že tato knihovna reprezentuje hodnotami pouze monomorfní typy, a dále, že typový systém Haskellu nepracuje s podtypy.

Při přidávání částečných typů do netypovaného jazyka

- je nutno pro jazyk vybudovat celý typový systém,
- sestrojit korektní algoritmus pro otypování a implementovat ho buď do kompilátoru či interpretu jazyka, anebo pro otypování (typovou kontrolu) vytvořit separátní nástroj,
- je třeba rozšířit gramatiku jazyka o typové anotace a modifikovat syntaktický analyzátor kompilátoru/interpretu,
- postarat se o ošetření typových chyb v době překladu,
- též bývá vhodné vymazat nepotřebné běhové příznaky a zrušit testy při aplikaci funkce na hodnotu bez příznaků.

Centrální roli mezi typy pro částečné typování hraje zvláštní typ *Dynamic*, jímž se otypují hodnoty, pro něž nelze v době překladu dostatečně přesný typ odvodit a pro něž se pouze generují běhové kontroly. Typ *Dynamic* lze při odvozování nahradit libovolným typem a naopak, libovolný typ lze nahradit typem *Dynamic*. J. Siek v [23] popisuje způsob otypování termů a vysvětluje, proč je nutný převod oběma směry. Zavádí proto reflexivní a symetrickou relaci $\sim$ *typové konsistence*, která rozšiřuje rovnost na typech pravidlem $\alpha \sim Dynamic$ pro všechny typy α.

Částečné přidávání statických typů do netypovaných jazyků řeší nástroje vyjmenované v sekci 4.

4 Existující systémy pro dynamické jazyky

Existuje několik systémů zavádějících typy do netypovaných jazyků. Níže jsou uvedeny nástroje pro Javascript, PHP a Python. Kromě nich existuje několik dalších jazyků,[7] které v menší či větší míře počítají alespoň s budoucím rozšířením o částečné

[7] Například Ruby nebo TypeScript.

typování — například možností typových anotací, byť je zatím ignorují a zacházejí s nimi jen jako s komentáři.

4.1 Flow pro Javascript

Nástroj *Flow* [5] zavádí typový systém, který přidává do Javascriptu statické typy a syntakticky ho rozšiřuje. Nejvýznamnější ze syntaktických extenzí je možnost přidávat do kódu typové anotace. Flow provádí typovou kontrolu a upozorňuje uživatele na zjištěné typové nekompatibility. Otypování je částečné, v přeloženém kódu se ponechávají běhové příznaky.

Flow navíc poskytuje některé další pomocné nástroje, například *lint* pro kontrolu stylu a lexikální kontrolu apod.

Jeho typový systém podporuje:

- typy funkcí (např. `len : (string) => number`),
- součinové typy,
- podtypy a průnikové typy,
- zavádí sadu typových konstruktorů (např. `?`, odpovídající haskellovému *Maybe*),
- definice vlastních typových funkcí (pouze druhu $\star \to \star$),
- u typových funkcí lze stanovit varianci v předepsaných argumentech.

Flow je poměrně mladý nástroj a stále se vyvíjí. Autorský tým o něm publikuje informace pouze na svých webových stránkách [5].

4.2 Hack pro PHP

Hack je svými autory popisován jako typovaný programovací jazyk založený na jazyku PHP a určený pro virtuální stroj HHVM [2]. Je vyvíjen od roku 2014 týmem autorů ze společnosti Facebook. Ti si byli vědomi rozšířenosti jazyka PHP a navrhli typovaný jazyk, který se jazyku PHP velmi přibližuje. Má s ním natolik velký průnik, že je označován za jeho dialekt a za jazyk určený pro programátory PHP. Nicméně existují některé prvky jazyka PHP, které Hack záměrně nepodporuje.[6]

Typový systém jazyka Hack obsahuje zejména:

- typy jednoduchých prvořádových funkcí,
- součinové typy,
- řadu zabudovaných typových funkcí pro běžné datové struktury (pole, tabulky, posloupnosti, množiny),

- řadu speciálních typových konstruktorů např. pro výjimky, synchronní a asynchronní funkce apod.

Podobně jako Flow, typový systém Hacku je *částečný*: pro podvýrazy, které nejsou dostatečně otypovatelné otypovávacím algoritmem Hacku, se generuje kód, který se spoléhá na běhové příznaky u hodnot a při nesouladu způsobí běhovou chybu. Hack je tedy kombinací typovaného a netypovaného jazyka.

Autoři publikovali informace o Hacku pouze na webových stránkách [21]. Je to hlavně proto, že Hack a jeho typový systém jsou stále ve vývoji a některé věci ještě nejsou domyšlené. Na úrovni statického typování je Hack podobný nástroji Flow.

4.3 MyPy pro Python

Python je sice netypovaný (dynamický) jazyk, ale už jeho verze 3 přidala do syntaxe typové anotace, obsažené v Návrhu rozšíření jazyka PEP-484 [22]. S těmi se však normálně zachází jen jako s komentáři a standardním interpretem jsou ignorovány. Rozšíření o anotace bylo zamýšleno pro využití pozdějšími nástroji pro otypování nebo jinou formu statické analýzy.

V roce 2014 se objevil nástroj *MyPy* [3]. Zavádí typový systém pro Python a využívá anotací k typové kontrole.

Typový systém MyPy poskytuje

- typy funkcí,
- podtypy a typy sjednocení,
- součinové typy,
- typy pro třídy a objekty,
- několik zabudovaných typových konstant a typových funkcí,
- omezenou formu parametrického polymorfismu (typový konstruktor *Generic*).

Typová kontrola je však, podobně jako u Hacku nebo u Flow, částečná — zůstává dynamické testování běhových příznaků. Typový systém, jejž MyPy poskytuje, je tedy dalším příkladem systému s částečným typováním. Uživatel může i explicitně anotovat proměnnou nebo konstantu zvláštním typem *Any*, který odpovídá největšímu typu $\top$ vzhledem k relaci $<:$. V případě takového explicitního otypování se generují běhové kontroly.

5 Směry a cíle další práce

Naše další práce bude směřovat ke studiu existujících a vývoji nových typových systémů pro třídu netypovaných, tzv. *dynamických* jazyků. Omezení na netypované jazyky je na první pohled bezobsažné a nijak zásadně neohraničuje studované typové systémy — triviální typový systém lze rozšířit jakýmkoliv způsobem a kterýmkoliv směrem. Avšak v praxi používané dynamické jazyky se vyznačují některými společnými vlastnostmi, které by měl nový typový kalkul reflektovat. Jsou to mimo jiné:

- Dynamické jazyky pracují s objekty nebo se záznamy. Typový kalkul by tedy měl podporovat součinové typy.
- Zacházejí s daty způsobem, pro jehož formální popis se hodí podtypový polymorfismus. Například je-li parametr metody objektem, pak její skutečný argument v aplikaci může obsahovat libovolná pole navíc.
- Odvozování typů v kalkulu s podtypy vede k potřebě průnikových typů a typů sjednocení.
- Dynamické jazyky často umožňují uživatelskou definici datových struktur a zacházejí s funkcemi nad těmito strukturami způsobem, pro jehož formální popis se hodí parametrický polymorfismus.

Tyto vlastnosti vedou ke kalkulům s nerozhodnutelným otypováním. I když nerozhodnutelnost otypování ještě nemusí být problémem, je vhodné udržet aspoň rozhodnutelnost typové kontroly. Pro typovou kontrolu je třeba vyžadovat explicitní typové anotace všech proměnných. Takový požadavek při psaní větších programů příliš nevadí, spíše naopak: explicitní anotace pomáhají důkladnému pochopení problému, a jsou tudíž žádoucí. Avšak na druhé straně, pro rychlé „skriptování“ jsou explicitní typové anotace pro autory programů přítěží a jejich vyžadování je pro ně nepříjemné.

Z těchto důvodů se slibnější jeví alternativní cesta, jíž je částečné typování (viz odst. 3.2). Touto cestou ostatně jdou nástroje popsané v sekci 4. Kalkul s částečným typováním umožní staticky otypovat všechny výrazy i za cenu, že součástí typu některých výrazů bude neučitý dynamický typ odkazující se na běhovou kontrolu.

Teorie typů se v uplynulých letech rozvíjela a přinesla četné výsledky. Otevírá i nové otázky a témata, které si zaslouží vyšetřit:

- rozhodnutelnost otypování navržených kalkulů a jejich extenzí;
- kombinace částečného typování s parametrickým polymorfismem;
- vztah relace typové konsistence a relace podtypu v kalkulech s částečným typováním;
- otypování příkazů imperativního jazyka s cílem vnést do jazyka referenční transparentnost;

- využití vyšších sort nebo typových tříd à la Haskell;
- vztah mezi součinem a součtem na jedné a průnikem a sjednocením na druhé straně;
- vztah mezi podtypy a parametrickým polymorfismem — kombinace podtypového a parametrického polymorfismu se v typových kalkulech popisovaných v literatuře nevyskytuje.

Jelikož se tato témata a problémy týkají vlastností typového kalkulu, který je naším cílem, bude součástí další práce i zkoumání zmíněných otázek.

Literatura

[1] The Glasgow Haskell compiler — State-of-the-art, open source compiler and interactive environment for Haskell.
URL `https://wiki.haskell.org/GHC`

[2] HipHop Virtual Machine Documentation.
URL `https://docs.hhvm.com/hhvm/`

[3] MyPy, experimental optional static type checker for Python.
URL `http://mypy-lang.org/`

[4] Haskell 2010 Language Report. 2010.
URL `https://www.haskell.org/definition/haskell2010.pdf`

[5] Flow documentation — Guides and references. 2014.
URL `https://flow.org/en/docs/`

[6] Hack – unsupported features. 2017.
URL `https://docs.hhvm.com/hack/unsupported/`

[7] Abadi, M.; Cardelli, L.: *A Theory of Objects*. Springer, 1996, ISBN 978-0-387-94775-4, 396 s.

[8] Barendregt, H. P.: Introduction to generalized type systems. *Journal of Functional Programming*, ročník 1, 1991: s. 124–154, ISSN 0956-7968.

[9] Barendregt, H. P.: *Lambda calculi with types*, ročník 2, kapitola 2. Clarendon Press, 1993, ISBN 978-0-198-53761-8.

[10] Barendregt, H. P.: *The lambda calculus, its syntax and semantics*. College Publications, 2012.

[11] Brady, E.: *Type-Driven Development with Idris*. Manning Publications, 2017, ISBN 978-1-617-29302-3.

[12] Chlipala, A.: *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013, ISBN 978-0-262-02665-9.

[13] Coquand, T.; Huet, G.: The calculus of constructions. Technická Zpráva 530, Institut National de Recherche en Informatique et en Automatique, Paris, 1986.

[14] Damas, L.; Milner, R.: Principal type-schemes for functional programs. In *9th Sigplan-Sigact symposium on Principles of programming languages*, ACM, 1982.

[15] Hall, C.; Hammond, K.; Jones, S. P.; aj.: Type classes in Haskell. *ACM Transactions on Programming Languages and Systems*, ročník 18, 1996: s. 241–256.

[16] Harper, R.: *Practical Foundations for Programming Languages*. 2013, ISBN 1-107-02957-0.

[17] Jones, S. P.; Jones, M.; Meijer, E.: Type Classes: An Exploration of the Design Space. In *In Haskell Workshop*, 1997.

[18] Milner, R.: A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, ročník 17, 1978: s. 348–375.

[19] Milner, R.; Tofte, M.; Harper, R.: *The definition of standard ML*. The MIT Press, 1997, ISBN 0-262-63181-4.

[20] Norell, U.; Chapman, J.: Dependently Typed Programming in Agda. In *Advanced Functional Programming. LNCS 5832*, Springer, 2009, s. 230–266.

[21] O'Sullivan, B.; Verlaguet, J.; Menghrajani, A.: Hack – Programming Productivity Without Breaking Things.
URL `https://hacklang.org/`

[22] van Rossum, G.; Lehtosalo, J.; Łukasz Langa: PEP 484 – Type hints for Python.
URL `https://www.python.org/dev/peps/pep-0484/`

[23] Siek, J. G.; Taha, W.: Gradual typing for functional languages. In *In Scheme and Functional Programming Workshop*, 2006, s. 81–92.

[24] Thatte, S. R.: Quasi-static typing. In *Proceedings of the 17th Symposium on Principles of Programming Languages (POPL)*, ACM, New York, 1990, s. 367–381.

[25] Wells, J. B.: Typability and type-checking in the second-order lambda calculus are equivalent and undecidable. In *Proceedings of the 9th Annual Symposium on Logic in Computer Science*, Paris: IEEE Computer Society Press, 1994.