Kalkuly zabývající se objekty

Jakub Černohorský
student na FIT VUT

text do předmětu Teorie programovacích jazyků
1. února 2003

Obsah

1 Úvod

Lambda kalkulus je jako nástroj použitelný pro popis jazyků. S objevením objektově orientovaných jazyků se hledala možnost uplatnit lambda kalkulus i v této oblasti. Vznikly objektové lambda kalkuly.

Je několik rozdělení pro ně, které bychom mohli použít. První z nich je rozdělení na na imperativní a funkcionální objektové kalkuly.

Dále můžeme dělit podle kritéria, jak se daný programovací jazyk dívá na objekty a třídy. Z tohoto pohledu se dá rozdělit množina objektově orientovaných jazyků na dvě velké skupiny. První skupina, kam se řadí *Simula, C++, Smalltalk, Eiffel, Java, . . .* , je nazývaná „class-based“. To znamená, že systém jazyka je založen na třídách a objektech, které jsou instancemi těchto tříd.

Druhá skupina je označována jako „object-based“. Do této skupiny se řadí jazyky *Self, Obliq a další*. U těchto jazyků je dědičnost implementována na úrovni objektů. Nový objekt je vytvořen jako kopie již existujícího objektu (prototypu).

Další výrazy spojující se s tímto rozdělením jsou „embedding-base“ a „ delegation-based“. Jedná se o popis jakým způsobem jsou metody předány objektům. V prvním případě jsou metody přímo vloženy do nově vytvářeného objektu, v druhém případě se odkazuje na prototyp.

V tomto dokumentu jsou zběžně popsány tři objektové kalkuly, z nichž pouze tomu prvnímu je věnována větší pozornost.

2 Lambda kalkulus pro objekty

2.1 Netypovaný lamda kalkulus s obekty

Dále popisovaný lamba kalkulus pro objekty s podporou specializace metod je uveden v [FHM94]. Jde o netypovaný lamda kalkulus se čtyřmi syntaktickými formami věnovanými objektům. Dědičnost je provedena pomocí delegace. Popis výrazů je následující:

$$e ::= x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \langle\rangle \mid e \Leftarrow m \mid \langle e_1 \leftarrowtail m = e_2 \rangle \mid \langle e_1 \leftarrow m = e_2 \rangle$$

V této gramatice mají symboly tyto významy: x je proměnná, c označuje konstantu, $\lambda\, x.e$ je lambda abstrakce a $e_1 e_2$ je aplikace. Jednotlivé objektové formy jsou popsány následujícně:

$\langle\rangle$	– prázdný objekt
$e \Leftarrow m$	– zaslání zprávy m objektu e
$\langle e_1 \leftarrowtail m = e_2 \rangle$	– rozšíření objektu e_1 novou metodou m s tělem e_2
$\langle e_1 \leftarrow m = e_2 \rangle$	– nahrazení těla metody m objektu e_1 tělem e_2

Poslední dvě objektové formy se liší pouze použitím. U $\langle e_1 \leftarrowtail m = e_2 \rangle$ se předpokládá použití pouze v situaci, kdy objekt nemá metodu m. Na druhou stranu použití $\langle e_1 \leftarrow m = e_2 \rangle$ je možné pouze v situaci kdy metoda m neexistuje.

2.2 Příklad

Pro ukázku definice objektů tímto způsobem uvedu příklady použité ve výše uvedené práci.

První příklad ukazuje jednoduchý záznam, který lze namodelovat pomocí kalkulu. V tomto příkladě se objekt skládá pouze s datových složek.

Zavedou se pouze zjednodušení pro definici metod. Původně by měla definice objektu následující tvar: $\langle \ldots \langle\rangle \leftarrowtail m_1 = e_1 \rangle \ldots \leftarrowtail m_k = e_k \rangle$. Pro zjednodušení se zavede tvar: $\langle m_1 = e_1, m_2 = e_2, \ldots, m_k = e_k \rangle$. Přičemž $m_1, \ldots, m_k$ jsou navzájem různá jména metod.

Vyhodnocení zaslání metody objektu se provede následujícím předpisem:

$$\langle m_1 = e_1, m_2 = e_2, \ldots, m_k = e_k \rangle \Leftarrow m_i \ \stackrel{eval}{\longrightarrow} \ e_i \langle m_1 = e_1, m_2 = e_2, \ldots, m_k = e_k \rangle$$

Tento způsob umožní vyhodnocení zaslání metody. Vezmeme tělo metody, jejíž méno jsme použili při zaslání zprávy a jako parametr zadáme objekt. Pro identifikaci daného objektu je přímo do lambda abstrakce přidán speciální symbol *self*. Tento symbol se často vyskytuje v objektově orientovaných jazycích pro identifikaci objektu, k němuž metoda patří (např. v C++ jde o klíčové slovo *this*).

Takže zpět k prvnímu příkladu. Jde o záznam s dvěmi komponenty. Modeluje bod se dvěmi souřadnicemi x a y.

$$x \stackrel{def}{=} \langle x = \lambda self.3, y = \lambda self.2 \rangle$$

Když teď máme nadefinován objekt, můžeme ukázat zavolání metody

$$x \Leftarrow r \ \stackrel{eval}{\longrightarrow} \ (\lambda self.3) r \ \stackrel{eval}{\longrightarrow} \ 3$$

Druhým krokem je zde β-redukce. Tímto způsobem lze namodelovat jakýkoliv záznam jehož metodami jsou konstantní funkce.

Dalším příkladem demonstrujícím metodu s parametrem je bod s jedním rozměrem a s definovanou metodou $move$. Objekt má pak následující tvar:

$$\begin{array}{lll} p \stackrel{def}{=} & \langle & x = \lambda self.3 \\ & & move = \lambda self.\lambda dx.\langle self \leftarrow x = \lambda s.(self \Leftarrow x) + dx \rangle \\ & \rangle & \end{array}$$

Metoda $move$ má následující chování. Pokud je aplikován na ní object a posunutí dx, provede nahrazení metody x objektu novou metodou, která má jinou hodnotu. Výpočetní posloupnost vypadá takto:

$$\begin{array}{rcl} p \Leftarrow move\ 2 & = & (\lambda self.\lambda x.\langle \ldots \rangle)\ p\ 2 \\ & = & \langle p \leftarrow x = \lambda s.(p \Leftarrow x) + 2 \rangle \\ & = & \langle p \leftarrow x = \lambda s.3 + 2 \rangle \\ & = & \langle p \leftarrow x = \lambda self.5 \rangle \end{array}$$

Z posledního řádku lze vidět, že vznikl nový objekt s předefinovanou metodou x, jejíž nová hodnota je 5.

Pokud se použije pravidlo pro rovnost objektů,

$$\langle \langle m_1 = e_1, \ldots, m_k = e_k \rangle \leftarrow m_i = e_i' \rangle \ = \ \langle m_1 = e_1, \ldots, m_i = e_i', \ldots, m_k = e_k \rangle$$

dostaneme nový objekt následujícího tvaru

$$\begin{array}{lll} p \Leftarrow move\ 2 \ = & \langle & x = \lambda self.5, \\ & & move = \lambda self.\lambda dx.\langle \ldots \rangle \\ & \rangle & \end{array}$$

Je vidět, že nový objekt je identický s původním s pouze s rozdílem, že metoda x vrací hodnotu 5.

Poslední použití příklad demonstruje dědičnost. Nový objekt bude reprezentovat bod, který kromě souřadnice bude obsahovat ještě barvu bodu.

$$cp \stackrel{def}{=} \langle p \leftarrowtail color = \lambda self.red \rangle$$

Pokud aplikujem metodu $move$ na barevný bod, dostaneme znovu barevný bod, ale se správnou aplikací metody $move$. Nový bod bude mít posunutu souřadnici x.

2.3 Vzájemná rekurze

Nastavený systém bude fungovat dobře do té doby, než se vyskytne požadavek na vytvoření navzájem rekurzívních funkcí. V tom případě nás totiž omezuje požadavek na to, že můžeme volat pouze ty metody, které již jsou nadefinovány. Mámé tedy dvě metody m a n, kdy metoda m volá metodu n a naopak. Toto omezení lze ale jednoduše obejít. Nejdříve vytvoříme prázdnou metodu m s prázdným tělem. Teď již můžeme vytvořit metodu n, protože metoda m již existuje. Posledním krokem, který je nutno provést, je nahrazení prázdné metody m. Nahradíme tedy tělo metody m tělem, které už správně volá existující metodu n.

2.4 Operační sémantika

První krok je definice pravidla pro výběr a aplikaci metody. Autoři kalkulu zvolili permutační pravidlo.

$$\langle\langle e_i \leftarrow n = e_2\rangle m = e_3\rangle \;=\; \langle\langle e_1 \leftarrow m = e_3\rangle n = e_2\rangle$$

Operace $\leftarrow$ může být jak přidání metody, tak její nahrazení. Použitím pravidla dosáhneme pohledu na objekt spíše jako na množinu metod než jako na jejich posloupnost. Toto je ale v rozporu s dříve zmiňovaným pravidlem, které říká, že metoda musí být nejdříve definována než může být použita. Proto musí být použita klička.

$$\langle\langle\langle m_1 = e_1, \ldots, m_k = e_k\rangle \leftarrow m_1 = e'_1\rangle \ldots \leftarrow m_k = e'_k\rangle$$

Je definována „standardní forma“, která nejdříve nadefinuje všechny metody s libovolnými těly. Následně můžeme aktualizovat metody v libovolném pořadí.

$$\begin{array}{lcl}
\langle\langle e_1 \leftarrow n = e_2\rangle \leftarrowtail m = e_3\rangle & \stackrel{book}{\longrightarrow} & \langle\langle e_1 \leftarrowtail m = e_3\rangle \leftarrow n = e_2\rangle \\
\langle\langle e_1 \leftarrow n = e_2\rangle \leftarrow m = e_3\rangle & \stackrel{book}{\longrightarrow} & \langle\langle e_1 \leftarrow m = e_3\rangle \leftarrow n = e_2\rangle \\
\langle e_1 \leftarrowtail m = e_3\rangle & \stackrel{book}{\longrightarrow} & \langle\langle e_1 \leftarrowtail m = e_3\rangle \leftarrow m = e_3\rangle \\
\langle\langle e_1 \leftarrow m = e_2\rangle \leftarrow m = e_3\rangle & \stackrel{book}{\longrightarrow} & \langle e_1 \leftarrow m = e_3\rangle \\
(\lambda x.e_1)e_2 & \stackrel{eval}{\longrightarrow} & [e_2/x]e_1 \\
\langle e_1 \leftarrow m = e_2\rangle \Leftarrow m & \stackrel{eval}{\longrightarrow} & e_2\langle e_1 \leftarrow m = e_2\rangle
\end{array}$$

První čtyři pravidla uvedená v tabulce se týkají převodu do „standardní formy“. Tyto pravidla umožňují správnou permutaci metod k dosažení požadovaného stavu. Důležitá poznámka souvisí s typovým systémem (zmíněném v další kapitole), kdy použití pravidla $e \stackrel{book}{\longrightarrow} e'$ nemění typ objektu.

2.5 Typ třídy

Typ objektu autoři nazvali „třída“. Jak sami podotýkají, je zde samozřejmě problém s kolizí terminologií, kde třída značí jak rozhraní, tak typ objektu, ale přesto je třída obvyklý výraz pro typ objektu.

Typ objektu je definován takto:

$$\mathbf{class}\ t.\langle m_1 : \tau_1, \ldots, m_k : \tau_k \rangle$$

Tento zápis znamená,že objekt e má typ t a každé vyvolání metody $e \Leftarrow m_i$ má typ τ_i. Typ τ_i může v sobě obsahovat libovolný výskyt typu t. Tedy $\mathbf{class}\ t.\langle \ldots \rangle$ je rekurzívně definovaným typem.

Zaslání zprávy objektu odpovídá následujícímu výrazu

$$\frac{e : \mathbf{class}\ t.\langle \ldots m : \tau \rangle}{e \Leftarrow\ m\ : [\mathbf{class}\ t.\langle \ldots m : \tau \rangle / t]\tau}$$

kde substituce za t odpovídá rekurzívní definici typu. Pokud tedy aplikujeme definici na bod uvedený v oddíle s příklady, dostaneme:

$$\mathbf{class}\ t.\langle \mathrm{x} : int, \mathrm{move}\ int \to t \rangle$$

Metoda bodu $p \Leftarrow x$ vrací integer a metoda $p \Leftarrow move\ n$ vrací stejný typ jako má p.

Pro vysvětlení, jak se chovají metody při dědičnosti si vezmeme příklad s barevným bodem. Barevný bod má následující typ:

$$\mathbf{class}\ t.\langle \mathrm{x} : int, \mathrm{move} :\ int \to t, \mathrm{color} : colors \rangle$$

Jak je vidno už z prvního pohledu metoda move vrací typ t. Což už je v tomto případě typ, který reprezentuje barevný bod. Specializace probíha automaticky při dědění.

2.6 Typy, řádky a druhy

V tomto oddíle je popis všech typů.

Typy

$$\tau\ ::=\ t \mid \tau_1 \to \tau_2 \mid \mathbf{class}\ t.R$$

Řádky

$$R\ ::=\ r \mid \langle\rangle \mid \langle R \mid m : \tau \rangle \mid \lambda t.R \mid R\tau$$

Druhy

$$kind\ ::=\ T \mid \kappa$$
$$\kappa\ ::=\ T^n \to [m_1, \ldots, m_k]$$

Prostředí.

$$\Gamma\ ::=\ e \mid \Gamma, x : \tau \mid \Gamma, t : T \mid \Gamma, r : \kappa$$

Standardní správné formy:

$\Gamma \vdash *$	dobře vytvořený kontext
$\Gamma \vdash c : \tau$	typ termu
$\Gamma \vdash \tau : T$	dobře vytvořený typ
$\Gamma \vdash R : \kappa$	typ řádku

2.7 Typová pravidla

Pravidlo pro prázdný objekt.

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle \ : \ \mathbf{class}\ t.\langle\rangle}$$

Prázdný objekt nemá žádnou metodu a tudíž nemůže reagovat na žádnou zaslanou zprávu. Ale samozřejmě jde takovýto objekt rozšířit novými metodami. Pravidlo pro poslání metody objektu má tvar

$$\frac{\Gamma \vdash e : \mathbf{class}\ t.\langle R|m : \tau\rangle}{\Gamma \vdash e \Leftarrow m : [\mathbf{class}\ t.\langle R|m : \tau\rangle/t]\tau}$$

Poslední pravidlo, zde neuvedené, se vztahuje k přidání metod do objektu.

3 Rozšíření lambda kalkulu pro objekty o třídy

Lambda kalkulus popsaný v předchozím oddíle se zaobírá pouze objekty. V prácí [BF98] je rozšířen o možnost popsat jazyk, který obsahuje třídy („class-based“ jazyk).

V tomto případě je zvolen jiný přístup než použití premetod. V tomto klasickém přístupu je objekt A třídou. Obsahuje premetody a metodu new. Tato metoda, pokud je zavolána, vrací object do něhož jsou premetody „nainstalovány“ jako normální metody.

3.1 Rozšířitelný objektový model

Součástí tohoto modelu jsou dva principy: za prvé objektový systém, který podporuje dědičnost a zasílání zpráv a za druhé mechanismus zapouzdření, který zajišťuje ukrytí vnitřní struktury.

Vlastnosti tohoto systému jsou následující. Vytváří rozšiřitelné a soudržnou kolekci metod. Není možné u metod, které jsou navzájem rekurzivní, nahradit jednu metodu jinou, která by tuto soudržnost porušovala. V systému, který používá premetody, není tato vlastnost zajištěna.

Systém garantuje inicializaci. Systém rozšiřitelných objektů umožňuje dědění konstruktorů tříd a kódu zajišťujícího inicializaci. Vezmeme-li příklad bodů uvedený již výše, pak třída $ColorPoint$, která je potomkem třídy $Point$, může vyvolat konstruktur své nadtřídy.

Posledními dvěmi vlastnostmi jsou explicitní hierarchie typů a automatická propagace změn.

3.2 Kalkulus

Změna oproti předchozí definici spočívá v přechodu od funkcionálního přístupu k přístupu imperativnímu. Původní posílání zpráv postupně

$$((e \Leftarrow m_1(args_1)) \ldots) \Leftarrow m_k(args_k))$$

bylo nahrazeno možností zaslat témuž objektu více zpráv:

$$(e \Leftarrow m_1(args)); \ldots ; (e \Leftarrow m_k(args_k))$$

Výrazy jsou oproti výše definovanému kalkulu rozšířeny:

$$\begin{aligned} e \;::=\; & x \mid c \mid \lambda x.e \mid e_1 e_2 \mid \mathbf{let}\ x = e_1 \mathbf{in}\ e_2 \mid \\ & \langle\rangle \mid \\ & e \Leftarrow m \mid \langle e_1 \leftarrow m = e_2 \rangle \mid \langle e_1 \leftarrowtail m = e_2 \rangle \mid \\ & e.f \mid \langle e_1.f := e_2 \rangle \mid \langle e_1.f \leftarrowtail: f = e_2 \rangle \mid \\ & \{r <:_w R :: L = R_1, e\} \mid \\ & \mathbf{Abstype}\ r <:_w R :: L\ \mathbf{with}\ x : \tau\ \mathbf{is}\ e_1\ \mathbf{in}\ e_2 \mid \\ & \Lambda().e \mid e * () \end{aligned}$$

První dva řádky mají již známý význam. Předposlední řádek definuje datovou abstrakci r s operací x. Tato konstrukce naváže r na konkrétní řádek definovaný v e_1. Navíc jsou zde ještě výrazy pro přidání atributu do objektu a pro nahrazení atributu v objektu (jde o výrazy $\langle e_1.f := e_2 \rangle$ a $\langle e_1.f \leftarrowtail: f = e_2 \rangle$).

3.3 Typy

Syntaxe typů se také změní, protože se nám zde objevují **pro** objekty (neboli také prototypy) a normální **obj** objekty, které již nejsou dále rozšiřitelné.

$$\tau ::= unit \mid c \mid \tau_1 \to \tau_2 \mid \mathbf{pro}.R \mid \mathbf{obj}.R$$

Pravidlo pro vytvoření podtypů mezi jednotlivými typy objektů je následujícího tvaru:

$$\frac{\rho \vdash R_1 <:_B R_2}{\rho \vdash \mathbf{probj}.R_1 <: \mathbf{obj}.R_2}$$

Toto pravidlo je použitelné jak pro změnu rozšiřitelného objektu na nerozšiřitelný tak i pro závislost mezi měnitelnými objekty. Vytváření podtypů funguje na úrovni seznamu komponentů. Následující dvě pravidla se vztahují k vytváření podtypů. Nejdříve pravidlo „do šířky“ a pak „do hloubky“.

$$\frac{\begin{array}{c}\rho \vdash R_1 <:_B R_2 \\ \rho \vdash \langle R_1 | l : \tau \rangle :: L_1\end{array}}{\rho \vdash \langle R_1 | l : \tau \rangle <:_B R_2}$$

$$\frac{\begin{array}{c}\rho \vdash R_1 <:_B R_2 \quad \rho \vdash \tau_1 <: \tau_2 \\ \rho \vdash \langle R_1 | m : \tau_1 \rangle :: L_1 \quad i \in \{1, 2\}\end{array}}{\rho \vdash \langle R_1 | m : \tau_1 \rangle <:_{B+d} \langle R_2 | m : \tau_2 \rangle}$$

Druhé pravidlo je aplikovatelné pouze na metody. Protože atributy se můhou měnit, neaplikuje se na ně toto pravidlo pro zajištění bezchybovosti typového systému.

3.4 Příklad

Jako příklad vezmeme již použitý příklad bodu a barevného bodu.

$$
\begin{array}{l}
\textbf{Abstype}\ Point <:_w P_{pub} :: L_p \\
\quad \textbf{with newP}\ : int \to \textbf{pro}.Point \\
\quad\quad \textbf{is}\ \{Point <:_w P_{pub} :: L_p = P_{priv}, \mathrm{Imp_p}\} \\
\quad\quad\quad \textbf{in} \\
\quad\quad\quad\quad \textbf{Abstype}\ CPoint <:_w CP_{pub} :: L_{cp} \\
\quad\quad\quad\quad\quad \textbf{with newCP}\ : color \to int \to \textbf{pro}.CPoint \\
\quad\quad\quad\quad\quad\quad \vdots
\end{array}
$$

kde

$$
\begin{array}{rcl}
P_{pub} & \stackrel{def}{=} & \langle \mathbf{getX} : int, \mathbf{setX} : int \to unit \rangle \\
P_{priv} & \stackrel{def}{=} & \langle x : int, \mathbf{getX} : int, \mathbf{setX} : int \to unit \rangle \\
 & \vdots & \\
L_p & \stackrel{def}{=} & \{\mathbf{x}, \mathbf{getX}, \mathbf{setX}\}
\end{array}
$$

ještě chybí ukázat tvar konstruktorů:

$$
\begin{array}{rcl}
\mathrm{Imp_p} \stackrel{def}{=} \lambda ix.\langle\langle\ \langle\rangle & \leftarrowtail & : x = ix\rangle \\
 & \leftarrowtail & \mathbf{getX}\ =\ \Lambda().\lambda \mathbf{self}.(\mathbf{self}.x)\rangle \\
 & \leftarrowtail & \mathbf{setX}\ =\ \Lambda().\lambda \mathbf{self}.\lambda newX.\mathbf{self}.x := newX\rangle
\end{array}
$$

Konstruktor pro barevný bod využívá objektu vytvořeného konstruktorem pro normální bod. Výraz $\langle\rangle \leftarrowtail \ldots$ pro přidání první metody je nahrazen výrazem $\mathbf{newP}\ ix \leftarrowtail \ldots$. Po tomto následuje přidání metod pro práci s barvou bodu. Rozdíl je také mezi objekty, které vrátí konstruktor pro bod a barevný bod. Normální bod vrací objekt, který je ještě dále rozšiřitelný narozdíl od konstruktoru pro barevný bod, který vrací „final“ objekt.

3.5 Mixins

Mixins jsou funkce, které umožní rozšířit objekt o metodu. Tato funkce dostane jako parametr objekt a vrací objekt nový, který obsahuje přidanou metodu. Uvedený příklad přidává do objektu typu $Point$ novou metodu **move**.

$$
\begin{array}{l}
\textbf{let addMove} = \Lambda().\lambda ob.\langle ob \leftarrowtail \mathbf{move} = \Lambda().\lambda s.\lambda dx.s \Leftarrow \mathbf{setX}(dx + s \Leftarrow \mathbf{getX})\rangle \\
\quad \textbf{in let}\ y = \langle\langle\rangle \leftarrowtail x = 3, \\
\qquad\qquad \leftarrowtail \mathbf{getX}\ = \Lambda().\lambda s.s.x, \\
\qquad\qquad \leftarrowtail \mathbf{setX}\ = \Lambda().\lambda s.\lambda nx.\langle s.x := nx\rangle\rangle \\
\quad\quad \textbf{in addMove}\ *\ ()y
\end{array}
$$

3.6 Operační sémantika

Operační sémantika je postavena pouze na redukci. Operační sémantika tohoto kalkulu je inspirována operační sémantikou kalkulu zmíněného v poslední kapitole tohoto textu. Narozdíl ale od kalkulu autorů Abadiho a Cardelliho, který obsahuje pouze objekty, tento kalkulus obsahuje také lambda abstrakce a polymorfické funkce.

Sémantika je založena na prostředích (*enviroments*) a uložení (*stores*). Prostředí E asociuje proměnnou termu s adresou. Uložení S asociuje adresy s hodnotami. Operační sémantika je vyjádřena redukčními pravidly následujícího tvaru:

$$\{S; E\} \models e \longrightarrow a \bullet S'$$

Tento zápis vyjadřuje výraz e je vyhodnocen na adrese a v uložení S a prostředím E. Po provedení je uložení S změněno na uložení S'. Hodnota asociována s výrazem e je uložena do S'.

Vyhodnocení $\leftarrow$ (nebo také $\leftarrowtail$) bylo již ukazáno dříve. Symbol η vyjadřuje konečné zobrazení z množiny adres do množiny hodnot. Zobrazení $\eta[l = a]$ je stejné jako zobrazení η pouze s tím rozdílem, že l zobrazuje na a. Tedy můžeme nadefinovat pravidlo pro předefinování nebo přidání metody či atributu (symbol zahrnující toto vše mějme $\leftarrow\!\!\!\rho$).

$$\frac{\begin{array}{c}\{S; E\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ \{S_1; E\} \models e_2 \longrightarrow a_2 \bullet S_2 \\ S_2(a_1) = \eta \qquad a_3 \notin dom(S_2)\end{array}}{\{S; E\} \models \langle e_1 \leftarrow\!\!\!\rho\ l = e_2\rangle \longrightarrow a_3 \bullet (S_2, a_3 \rightarrowtail \eta[l = a_2])}$$

Výsledkem je pak modifikovaný objekt uložený na adrese a_3. Dalším pravidlem je pravidlo , které popisuje zavolání metody objektu.

$$\frac{\begin{array}{c}\{S; E\} \models e_1 \longrightarrow a_1 \bullet S_1 \\ S_1(a_1) = \eta \qquad\qquad \eta(m) = a_1' \\ S_1(a_1') = \langle \Lambda().e_1, E_1\rangle \\ \{S_1; E_1\} \models e_1 \longrightarrow a_2 \bullet S_2 \\ S_2(a_2) = \langle \lambda x.e_2, E_2\rangle \qquad x \notin dom(E_2)\end{array}}{\{S_2; E_2, x \rightarrowtail a_1\} \models e_2 \longrightarrow a_3 \bullet S_3}$$
$$\{S; E\} \models e \Leftarrow m \longrightarrow a_3 \bullet S_3$$

Nejdříve se vyhodnotí objekt na adrese a_1 k získání zobrazení η, dostaneme seznam metod a atributů asociovaných s objektem e. Následně dojde k vyhodnocení a získání těla metody $\lambda x.e_2$. Následně vyhodnotíme e_2 s navázáním x na adresu objektu, který je příjemcem zprávy. Tato vazba, kdy první parametr je navázán na objekt, jehož metoda se volá, zastupuje *self*.

4 Core kalkulus

Core kalkulus prezentovaný v [BPSM99] je jednoduchý kalkulus založený na třídách. Hlavní cíle návrhu tohoto kalkulu byly zapozdření dat, typicky prezentované v jazycích klíčovými slovy `private` a `protected`.

Další vlastností je strukturální vytvoření typu. Jde o to, že typ není dán třídou, které je instancí, ale pouze seznamem jmen a typů metod a atributů. Toto je rozdíl například od C++, kde je typ dán pouze příslušností k nějaké třídě.

Modulární výstavba objektu je vlastnost, která zajišťuje, že změna v implementaci nadtřídy nezpůsobí nutnost změny implementace podtřídy. Nadtřída se sama stará o inicializaci svých atributů.

4.1 Třídy a objekty

Objekty jsou záznamy jejichž položkami jsou metody. Metody jsou funkce, které mají vazbu na $self$ a privátní atribut.

Nové prvky, které jsou zavedeny jsou: *class values*, které reprezentují kompletní třídy, *class extension expressions*, které obsahují definice metod, atributů a konstruktorů a na závěr *instantiation expressions*, které reprezentují vytvoření objektů ze tříd.

4.2 Příklad

```
let EncryptedFile = extend File with
  protected decrypt = λkey.λself.λdata....;
  protected encrypt = λkey.λself.λdata....;
  method read = λnext.λkey.λself.λnbytes.self.decrypt(next(nbytes));
  method write = λnext.λkey.λself.λdata.next(self.encrypt(data))
  constructor λ(key,filename).{fieldinit = key, superinit = filename}
end in ...
```

Každá třída je definována jako rozšíření nějaké již existující třídy. Na vrcholu hierarchie stojí prázdná třída. Jak je vidět z příkladu, *class extension expressions* obsahují pouze veřejné (public) a chráněné (protected) metody a konstruktory. Místo deklarací polí obsahuje každá metoda λ-vazbu na jednu privátní položku ($\lambda key. \dots$).

Metody přistupují k objektu, k němuž patří pomocí vazby na *self*. Těla funkcí nadtřídy jsou přístupná pře parametr *next*. Konstruktor je jednoduchá funkce, která vrací záznam skládající se ze dvou součástí. První část je počáteční hodnota toho jediného privátního atributu a druhá část je hodnota, která je předána konstruktoru nadtřídy.

4.3 Syntaxe core kalkulu

Výraz $\mathbf{class}\langle v_g, [m_i]^{i \in Pub}, [m_j]^{j \in Prot} \rangle$ je výsledekem rozšířením třídy pomocí **extend**. Je to trojice, která obsahuje jednu funkci a dvě množiny porměnných. Funkce v_g je generátorem třídy, množina $[m_i]$ obsahuje veřejné metody třídy a $[m_j]$ obsahuje chráněné metody.

extend e_s **with**
  method $m_i = v_{m_i};^{i \in Pub}$
  protected $m_j = v_{m_j};^{j \in Prot}$
  constructor v_c;
end

V tomto výrazu zastupuje e_s nadtřídu a na dalších dvou řádcích jsou deklarace veřejných a chráněných metod třídy. Každé tělo metody v_{m_i} obsahuje **self** a pokud se jedná o předefinování metody nacházející se v nadtřídě, pak také **next**, která je navázána na metodu nadtřídy.

Konstrukce **new** e použije generátor v_g k vytvoření funkce, která vrací nový objekt. Je zde ještě nutno definovat kořenovou třídu, která se stane nadtřídou, všech nově tvořených tříd.

$$\mathbf{Object} \triangleq \mathbf{class}\langle \lambda \dots \lambda \dots . \{\}, [], [] \rangle$$

Kořenová třída je nutná pro zajištění stejného přístupu ke všem třídám. Všechny třídy, které nemají explicitně definovánu svoji nadtřídu jsou implicitně potomky třídy **Object**.

5 Imperativní objektový kalkulus

Tento objektový kalkulus byl představen v práci [AC96]. Jde o jednoduchý imperativní objektový kalkulus.

5.1 Syntaxe a neformální sémantika

Vyhodnocování termů je založeno na imperativní operační sématice se zpracováním deterministicky zleva doprava.

$a, b ::=$	term
x	proměnná
$[l_i = \sigma(x_i)b_i^{i\in 1\ldots n}]$	objekt (l_i jsou jednoznačná)
$a.l$	vyvolání metody
$a.l \Leftarrow (y, z = c)\sigma(x)b$	aktualizace metody
$clone(a)$	klonování

Jednoduchá definice objektu jako se soubor komponent $l_i = \sigma(x_i)b_i$. Každý název l_i je jedinečný a je asociován s metodou $\sigma(x_i)b_i$;. Pořadí těchto komponent nehraje roli.

Při vyhodnonocení volání metody $a.l$, se vyhodnotí nejdříve a a pak tělo metody s názvem l s tím, že hodnota a bude navázána na proměnnou self této metody.

Operace klonování $clone(a)$ vytvoří nový objekt, který ma všechny názvy stejné jako a a sdílí metody společně s a.

Konstrukce aktualizace metody je jasně vidělná, pokud použijeme jednodušší příklad $a.l \Leftarrow \sigma(x)(b)$. Tato konstrukce jednoduše vyhodnotí a a pak nahradí metodu s názvem l novou metodou $\sigma(x)(b)$ a vrátí nový objekt. Forma uvedená v tabulce výše je rozšířena o možnost vyhodnotit a a c pře provedením aktualizace. Vyhodnocené hodnoty jsou pak použity při samotné aktualizaci.

V netypovaném lambda kalkulu může být vyjádřena aktualizace metody pomocí let jako $let\ y = a\ in\ let\ z = c\ in\ y.l\ \Leftarrow \sigma(x)b$.

5.2 Atributy

Atributy v tomto kalkulu nejsou jinak definovany než metody. Řečeno jinak, všechny komponenty objektu metody. Zjednodušený zápis je $[l_i = b_i]$ pro atributy. Pro aktualizaci atributu napíšeme $a.l := b$ a pro přístup k atributu stejně jako pro metodu $a.l$. Vnitřně mezi metodami a atributy není rozdíl. Například aktualizace atributu je definavána jako

$$a.l := b \triangleq a.l \Leftarrow (y, z = b)\sigma(x)z \ \ pro\ \ y \notin FV(b)$$

s podmínkou, že $x,\ y,\ z$ jsou navzájem různé.

Reference

[FHM94] K.Fisher, J.Honsell, J.C.Mitchel: A lambda calculus of objects and method specialization. *Nordig J. Computing*, 1994.

[BF98] V.Bono, K.Fisher: An imperative, First-Order calculus with Object Extension. 1998.

[AC96] M.Abadi, L.Cardelli: An imperative object calculus. *Theory and Practice of Object Systems*, 1996.

[AC94] M.Abadi, L.Cardelli: A Theory of Primitive Objects, Untyped and First-Order Systems, 1994.

[Mor96] F.Morgan: Implementation of an Imperative Object Calculus. *Computer Science Tripos, Part II*, 1996.

[BPSM99] V.Bono, A.Patel, V. Shmatikov, J.Mitchell: A Core Calculus of Classes and Objects. *Electronic Notes in Theoretical Computer Science 20*, 1999.