Programovací jazyk a aplikační prostředí Self

Martin Švec

7. února 2004

1 Úvod

Programovací jazyk Self byl vyvinut ve spolupráci firmy Sun Microsystems a Stanfordské univerzity jako další z řady experimentálních jazyků postavených na objektovém paradigmatu. Jazyk Self do značné míry vychází z klasického jazyka Smalltalk, a to zejména celkovou koncepcí systému i syntaxí jazyka. Oproti Smalltalku se však snaží dosáhnout maximální možné čistoty a jednoduchosti objektově orientovaného principu. V jazyce Self neexistují žádné třídy objektů, nerozlišují se členské proměnné objektů a metody, neexistuje žádná explicitní specifikace dědičnosti na úrovni tříd a řada dalších konceptů známých z jiných objektově orientovaných jazyků. Ve zkratce se dá říci, že jazyk Self zná jen dva základní pojmy: objekt a slot. Objektem se rozumí unikátní entita obsahující množinu slotů, slot lze charakterizovat jako pojmenovanou referenci na jiný objekt. Na bázi objektů a slotů jsou pak realizovány všechny známé principy objektového paradigmatu.

Self, stejně jako Smalltalk, nezahrnuje pouze programovací jazyk jako takový, ale celé nezávislé aplikační prostředí. Toto prostředí obsahuje na nejnižší úrovni virtuální stroj, interpretující platformně nezávislý bytecode Selfu. Nad virtuálním strojem pak leží to, co sami autoři nazývají Self *world*, svět Selfu. Tento nezávislý svět je tvořený množstvím objektů, od prototypů základních datových struktur jako jsou např. seznamy a asociativní pole, přes objekty grafického uživatelského rozhraní až po celé aplikace. Stejně jako u Smalltalku se celé prostředí (objekty i přeložený bytecode) ukládá do jednoho souboru (snapshotu), umožňujícího návrat do identického stavu systému, jako byl v okamžiku uložení snapshotu.

2 Programovací jazyk Self

Tento oddíl popisuje základní syntaxi a sémantiku jazyka Self.

2.1 Objekty

Jak již bylo řečeno v úvodu, *objekt* je fundamentální (a vedle slotů i jedinou) konstrukcí jazyka Self. Doslova všechno v Selfu je objekt. Objektem jsou číslice, složené datové struktury, ale i kód programu nebo realizace řídicích struktur programu. Obecně je objekt tvořen množinou *slotů* a případně kódem. Každý slot je dvojice tvořená unikátním jménem a referencí na jiný objekt.

Přestože všechno v SELFu je objekt, ne všechny objekty mají stejný charakter. Principiálně lze rozlišit tři kategorie objektů: běžné datové objekty a dva typy objektů nesoucích kód: metody a bloky.

2.1.1 Syntaxe

Popis každého objektu je uzavřen v kulatých závorkách. Uvnitř popisu je ve svislých čarách uzavřena definice slotů a za ní může následovat kód. Sloty se oddělují tečkou. Příklady:

```
()
( |slot1. slot2.| )
( |slot3| 'Hello World' printLine )
```

První příklad definuje prázdný objekt, druhý příklad popisuje objekt bez kódu se dvěma neinicializovanými sloty `slot1` a `slot2`, třetí příklad definuje objekt s jedním slotem a kódem `'Hello World' printLine`.

2.1.2 Metody

Metody jsou objekty obsahující sloty formálních parametrů a kód. Deklarace slotů formálních parametrů začínají vždy dvojtečkou. Kód je tvořen sekvencí výrazů oddělených tečkami. Za návratovou hodnotu metody je automaticky považován výsledek posledního výrazu nebo objekt vrácený operátorem `^`. Kromě slotů uvedených v definici objektu obsahují metody navíc ještě implicitní slot `:self*`. Například

```
( | :x. :y | (x*x) + (y*y) )
```

definuje metodu se dvěma parametry, `x` a `y`, která po vyhodnocení vrací objekt nesoucí hodnotu $x^2 + y^2$.

Metoda může kromě slotů parametrů obsahovat i libovolný počet běžných slotů. Ty lze použít jako „lokální proměnné“ metody. Tedy

```
( | :x. :y. x2. y2| x2: x*x. y2: y*y. x2 + y2. )
```

definuje stenou metodu jako v předchozím případě, která však (přiznejme, že zcela zbytečně, pouze pro ilustrační účely) mezivýsledky `x*x` a `y*y` ukládá do slotů `x2` a `y2`.

Celý princip volání a vyhodnocování kódu metod objektů bude popsán později, v oddílu 2.4.

2.1.3 Bloky

Bloky jsou speciální objekty určené pro implementaci uživatelských řídicích struktur v kódu. Principiálně jsou ekvivalentní blokům známým z jazyka Smalltalk. Blok je syntakticky totožný s objektem metody, pouze je uzavřen v hranatých závorkách místo v kulatých. Na rozdíl od metody objekt bloku neobsahuje slot `:self*`, ale implicitní slot `parent*`, dále anonymní (nepřístupný) slot ukazující na aktivační objekt metody, v které byl blok aktivován a slot `value[:[With:[With:[...With:]]]]` obsahující anonymní objekt metody s vlastním kódem bloku.

Protože celý princip tvorby a aktivace bloků je z důvodů respektování kontextu, v kterém je blok volán, složitější než u metod, na tomto místě uvedeme pouze příklad použití bloku uvnitř metody:

```
( | :x. :y | x = y ifTrue: [^y] False: [^x + 1] )
```

Pokud $x = y$, metoda vrací y, jinak vrací $x + 1$. Metoda obsahuje dva bloky, `[^y]` a `[^x + 1]`, implementující větve podmíněného příkazu.

2.2 Sloty

V předchozím popisu syntaxe objektů se objevilo několik různých deklarací slotů. V tomto oddílu tedy shrneme všechna možná označení slotů a jejich význam.

2.2.1 Sloty určené pouze pro čtení

Sloty inicializované v definici objektu operátorem `=` jsou určené pouze pro čtení. Například zápis

```
( | x = 3 + 4 | )
```

vytváří objekt se slotem `x` obsahujícím referenci na objekt `7`. Tuto referenci už není možné žádným způsobem změnit.

Výraz za operátorem `=` je vyhodnocován v kořenovém kontextu (kontextu objektu `lobby`, který bude popsán v oddílu 3.1).

2.2.2 Sloty určené pro čtení i zápis

Sloty neinicializované nebo inicializované operátorem `<-` jsou určené pro čtení i zápis. Mějme například objekt

```
( | x <- 3 + 4 | )
```

Přestože syntakticky je deklarován pouze jeden slot, v objektu fyzicky vznikají sloty dva. Prvním slotem je `x`, obsahující referenci na `7`. Druhým slotem je `x:`, který ukazuje na tzv. *přiřazovací primitivum* (assignment primitive). Přiřazovací primitivum je speciální metoda poskytovaná virtuálním strojem, která do slotu `x` uloží referenci na svůj parametr. Přiřazení objektu `17` do slotu `x`,

```
x: 17
```

tedy zavolá přiřazovací primitivum uložené ve slotu `x:` s argumentem `17`, které uloží do slotu `x` referenci na objekt `17`.

Poznamenejme, že automaticky generovaný přiřazovací slot `x:` je obyčejný klíčovaný slot s jedním parametrem (viz oddíl 2.2.3). Vskutku, tento slot je možné odstranit nebo do něj přiřadit jinou metodu; v obou případech se pak ze slotu `x` stává slot určený poze ke čtení.

Jako sloty určené ke čtení i zápisu jsou vytvářeny i neinicializované sloty. Ty jsou implicitně inicializovány na objekt `nil`. Následující deklarace jsou tedy ekvivalentní:

```
( | x | )
( | x <- nil | )
```

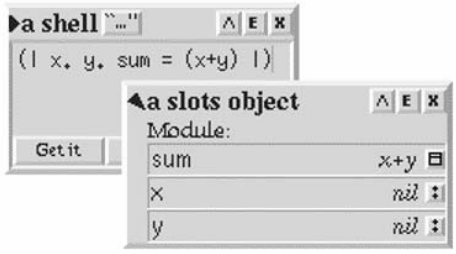
2.2.3 Sloty obsahující metody

Je-li do slotu určeného pouze ke čtení přiřazen objekt obsahující kód, slot se stává slotem metody. V tomto okamžiku je také do metody přidán slot `:self*`. Podle počtu parametrů se rozlišují unární sloty, binární sloty a klíčované sloty (keyword slots).

Unární slot obsahuje metodu bez parametrů (resp. jediným implicitním parametrem `:self*` je objekt, v kterém je metoda volána). Například

```
( | x. y. sum = (x + y) | )
```

definuje objekt se dvěma datovými sloty `x` a `y` a unárním slotem `sum` obsahujícím metodu `(x + y)`.



Obrázek 1: objekt s metodou sum

Binární slot je slot, jehož název obsahuje symboly operátorů a metoda očekává (kromě implicitního parametru `:self*`) jeden další parametr. Příkladem může být binární slot, který definuje pro objekt reprezentující komplexní číslo operaci sčítání:

```
( | re. im.
      + = ( |:arg| (clone re: re + arg re) im: im + arg im)
| )
```

Při deklaraci binárního slotu lze použít i alternativní syntaxe,

```
( | re. im.
      + arg = ((clone re: re + arg re) im: im + arg im)
| )
```

kdy je parametr metody uveden za názvem slotu.

Název *klíčovaného slotu* je složen z identifikátorů, oddělených dvojtečkami. Pro každý z identifikátorů pak musí být v metodě slot parametru. Příklad:

```
( | re. im.
      addRe:Im: = ( |:xre :xim| (clone re: re + xre) im: im + xim)
| )
```

definuje objekt s datovými sloty `re`, `im` a metodou `addRe:Im:` očekávající dva parametery, `:xre` a `:xim`. Opět zde existuje možnost alternativní syntaxe, kdy se parametry zapisují do názvu slotu:

```
( | re. im.
       addRe: xre Im: xim = ((clone re: re + xre) im: im + xim)
| )
```

2.2.4 Sloty formálních parametrů

Jak již bylo dříve uvedeno a několikrát demonstrováno v příkladech, sloty začínající dvojtečkou reprezentují *formální parametry* metod a bloků. Tyto sloty jsou při deklaraci vždy neinicializované, k inicializaci dochází až přiřazením skutečných parametrů při aktivaci metody či bloku (viz oddíl 2.4).

2.2.5 Rodičovské sloty

Slot, jehož název je následován hvězdičkou, se nazývá *rodičovský slot*. Rodičovské sloty jsou normální sloty ukazující na datové objekty, které však hrají důležitou roli v procesu vyhodnocování volání metod objektu. Pokud totiž při volání metody nenalezne SELF slot s odpovídajícím názvem v daném objektu, pokračuje v hledání slotu v objektech, na které ukazují rodičovské sloty (pro podrobný popis tohoto procesu viz oddíly 2.4 a 2.4.1). Rodičovskými sloty je tedy realizována v SELFu dědičnost, resp. vícenásobná dědičnost, je-li v objektu více slotů rodičovského typu. Pokud je rodičovský slot vytvořen jako slot pro čtení i zápis, objekt používá dynamickou dědičnost (je možné změnit rodiče v průběhu existence objektu).

2.2.6 Poznámkové sloty

Tyto speciální sloty umožňují k objektům a slotům přiřazovat poznámky. Uvádějí se ve složených závorkách, lze do nich přiřadit pouze řetězce a kromě dokumentace kódu nemají další význam:

```
( | { 'The following slot contains nil' someSlot = nil } | )
```

2.3 Výrazy

V oddílu 2.1.2 bylo řečeno, že každý kód v jazyce SELF je tvořen sekvencí *výrazů*. Každý výraz je buď objekt, nebo *zpráva* zaslaná *cílovému objektu* (receiveru). Syntaxe zasílání zpráv je shodná se syntaxí jazyka Smalltalk. SELF rozlišuje tři typy zpráv: unární zprávy, binární zprávy a klíčované zprávy. Výsledkem každého výrazu je reference na některý objekt.

2.3.1 Unární zprávy

Unární zpráva zaslaná objektu nemá žádné další parametry. Stejně jako ve Smalltalku se zapisuje jako cílový objekt následovaný tzv. *selektorem* zprávy. Mějme například objekt `11`. Pak zápis

```
11 factorial
```

znamená zaslání zprávy `factorial` objektu `11`. Jak bude podrobně popsáno níže, zaslání zprávy způsobí vyhledání unárního slotu s názvem `factorial` a vyhodnocení objektu v něm uloženého.

Unární zprávy jsou levě asociativní, tedy následující dva výrazy jsou ekvivalentní:

```
11 factorial print
((11 factorial) print)
```

V obou případech je zaslána zpráva `factorial` objektu `11`, jejímž výsledkem je objekt `39916800`. Tomu je následně zaslána zpráva `print`, která vypíše číslo `39916800` na standardní výstup.

Unární zprávy mají ze všech typů zpráv nejvyšší prioritu.

2.3.2 Binární zprávy

Binární zprávu tvoří cílový objekt, binární operátor (selektor) a argument zprávy. Například

```
11 + 43
```

znamená, že objektu `11` je zaslána zpráva `+` s parametrem `43`. Virtuální stroj v tomto případě vyhledá binární slot `+` a vyhodnotí v něm uložený objekt s parameterm `43`.

Binární zprávy se stejným operátorem jsou levě asociativní. Mezi binárními zprávami s různými operátory není asociativita definována a je nutné použít závorky. Tedy například zápisy

```
11 + 43 + 18 + 3.
((11 + 43) + 18) + 3.
```

jsou korektní a ekvivalentní, avšak výrazy

```
5 + 34 - 12 + 2.
11 + 4 * 7.
```

nejsou přípustné a je nutné je přepsat například takto:

```
5 + (34 - 12) + 2.
11 + (4 * 7).
```

Binární zprávy mají nižší prioritu než zprávy unární, ale vyšší prioritu než zprávy klíčované. Tedy výraz

```
5 + 3 factorial + pi sin.
```

je interpretován jako

```
(5 + (3 factorial)) + (pi sin).
```

2.3.3 Klíčované zprávy

Klíčovaná zpráva je zpráva o jednom a více parametrech zaslaná cílovému objektu. Synaxe je opět shodná se syntaxí jazyka Smalltalk, tedy například

```
5 min: 4 Max: 8.
```

je zpráva se selektorem `min:Max:` a s parametry `4` a `8` zaslaná objektu `5`.

Self vyžaduje, aby první klíčové slovo zprávy začínalo malým písmenem a ostatní klíčová slova zprávy začínala velkými písmeny. Zatímco

```
5 min: 4 Max: 7.
```

zasílá jednu zprávu `min:Max:` se dvěma parametry,

```
5 min: 4 max: 7.
```

je interpretováno jako dvě zprávy s jedním parametrem:

```
5 min: (4 max: 7).
```

Zprávy začínající znakem `_` (podtržítko) jsou vyhrazeny pro *primitiva virtuálního stroje* (speciální metody implementované přímo virtuálním strojem, jako je například přiřazovací primitivum).

Klíčované zprávy jsou pravě asociativní a mají nejnižší prioritu.

2.3.4 Zprávy s implicitním cílovým objektem

Pro úplnost dodejme, že Self připouští i tzv. *zprávy s implicitním cílovým objektem*. Je-li vynechán při zápisu zprávy cílový objekt, je zpráva zaslána sama sobě, do aktivačního objektu metody (viz oddíl 2.4.2). Zápisy

```
factorial.
+ 12.
```

však nejsou ekvivalentní zápisům

```
self factorial.
self + 3.
```

protože hledání slotu zprávy s implicitním cílovým objektem nezačíná v objektu `self` (kontexu aktivace metody), ale přímo v aktivačním objektu. Z důvodu potenciálních problémů z toho plynoucích autoři Selfu používání zpráv s implicitním cílovým objektem v běžném kódu nedoporučují.

2.4 Vyhodnocování a provádění kódu

V tomto oddílu je podrobně popsán mechanismus, kterým SELF provádí vyhodnocování výrazů a zasílání zpráv.

Je-li výrazem objekt, výraz vrací přímo tento objekt. Například

```
17. 'Hello World'.
```

jsou dva výrazy, první vrací objekt (číslo) `17`, druhý vrací objekt (řetězec) `'Hello World'`.

Je-li výrazem zaslání zprávy objektu, virtuální stroj vyhodnotí výraz v následujících dvou krocích:

1. Vyhledání slotu zprávy – SELF hledá slot, jehož název odpovídá selektoru (tj. názvu) zprávy. Slot je hledán v cílovém objektu a v objektech, na které ukazují rodičovské sloty (viz 2.2.5).

2. Vyhodnocení objektu ve slotu – odkazuje-li slot zprávy na datový objekt, je výsledkem zprávy tento datový objekt. Odkazuje-li slot na objekt metody, dojde k její aktivaci.

V tomto procesu může dojít k následujícím selháním:

1. Není nalezen žádný slot, jehož název odpovídá selektoru zprávy. V tom případě SELF hlásí tzv. *lookup error*, chybu při hledání slotu.

2. Je nalezen více než jeden slot s odpovídajícím názvem. Tato situace opět vede k chybovému hlášení.

3. Počet parametrů zprávy neodpovídá parametrům očekávaným objektem ve slotu zprávy. Toto selhání je v praxi syntakticky vyloučeno, protože selektor zprávy (a tudíž i název hledaného slotu zprávy) přímo implikuje počet parametrů.

Je zřejmé, že SELF při vyhodnocování objektu ve slotu zprávy akceptuje jak objekty datové tak i objekty metod. To umožňuje efektivní skrývání implementace objektů a snadnou změnu chování beze změny rozhraní objektu.

V následujících dvou oddílech jsou přiblíženy algoritmy vyhledání slotu zprávy a aktivace objektu metody.

2.4.1 Hledání slotu zprávy (message lookup)

Hledání slotu zprávy probíhá následujícím algoritmem (převzato z [1]). Virtuální stroj, počínaje v cílovém objektu zprávy, prochází orientovaný graf dědičnosti (specifikovaný rodičovskými sloty v objektech) a hledá sloty s daným názvem. Graf dědičnosti může být tvořit libovolný (i cyklický) graf, je garantováno že žádný objekt nebude prohledáván dvakrát na jedné cestě grafem. Při porovnávání názvů slotů se ignorují speciální symboly identifikující sloty parametrů a rodičovské sloty (tj. např. selektoru `self` odpovídá i slot `:self*`).

Vstup algoritmu:

obj = prohledávaný objekt
sel = selektor zprávy
V = množina již prohledaných objektů na této cestě grafem

Výstup algoritmu:

M = množina nalezených slotů

Algoritmus:

function $lookup(obj, sel, V)$
begin
  if $obj \in V$ **then** $M := \emptyset$
  else begin
    $M := \{s \in slots(obj) : slot\_name(s) = sel\}$;
    if $M = \emptyset$ **then**
      $M := parent\_lookup(obj, sel, V)$;
  end;
  return M;
end;

kde funkce *parent_lookup* je definována jako

function $parent\_lookup(obj, sel, V)$
begin
  $P := \{object\_in\_slot(s) : s \in slots(obj)$ and $is\_parent\_slot(s)\}$;
  $M := \bigcup_{o \in P} lookup(o, sel, V \cup \{obj\})$;
  return M;
end;

Uvažujme například výraz

```
17 min: 3.
```

Tento výraz zasílá zprávu se selektorem `min:` do objektu `17`. Algoritums nejprve hledá slot s názvem `min:` v objektu `17`. Zde takový slot neexistuje, hledání tedy pokračuje v rodičovských slotech. Objekt `17` má jediný rodičovský slot `parent*`, který ukazuje na objekt `traits smallInt`.[1] V objektu `traits smallInt` opět slot `min:` neexistuje, avšak existuje zde další rodičovský slot `parent*` ukazující na objekt `traits integer`. V `traits integer` slot `min:` také není a jediným rodičovským slotem je `parent*` ukazující na objekt `traits number`. V tomto objektu pak algoritmus nalezne slot `min:` inicializovaný jako

```
min: x = ( < x ifTrue: self False: x)
```

Protože je to jediný nalezený slot a je inicializovaný na objekt metody, vyhledávání je úspěšně dokončeno a objekt metody může být aktivován.

[1] Objekt `traits smallInt` implementuje sdílené chování číselných objektů rozsahu malého integeru. Podrobnosti o objektech typu `traits` viz oddíl 3.1.4.

2.4.2 Aktivace metody

Uvažujme opět výraz

```
17 min: 3.
```

Cílovým objektem zprávy je `17` a ke zprávě byl nalezen odpovídající slot v objektu `traits number` ukazující na metodu

```
( |:x| < x ifTrue: self False: x)
```

Připomeňme, že objekt metody navíc ještě obsahuje automaticky generovaný slot `:self*`, takže fyzicky má celá metoda tvar

```
( |:self*. :x| < x ifTrue: self False: x)
```

V prvním kroku aktivace metody virtuální stroj SELFu vytvoří *kopii* objektu metody – vzniká nový objekt, tzv. *aktivační objekt metody*. Další činnost probíhá už pouze v rámci aktivačního objektu.

V druhém kroku aktivace je slot `:self*` aktivačního objektu metody inicializovaný na *cílový objekt zprávy*, v našem případě objekt `17`. Sloty parametrů aktivačního objektu metody jsou inicializovány na objekty předané jako parametry zprávy, v pořadí jak byly uvedeny. Konkrétně v tomto příkladu je do slotu `:x` umístěn odkaz na objekt `3`. Tím je vymezen tzv. *aktivační kontext*, aktivační objekt metody je navázán na cílový objekt a má přístup k objektům parametrů zprávy:

```
( |:self* = 17. :x = 3| < x ifTrue: self False: x)
```

Povšimněme si, že sloty aktivačního kontextu jsou incializovány jako sloty určené pouze pro čtení.

Třetím krokem aktivace je vyhodnocení všech výrazů v kódu aktivačního objektu, zcela stejným postupem jako byl popsán u vyhodnocování výrazu `17 min: 3`.

V našem příkladu se jako první začne vyhodnocovat podvýraz `< x`. Ten má další podvýraz, `x`, ke kterému je nalezen slot `:x` přímo v aktivačním objektu, ukazující na `3`. Vyhodnocením datového objektu je objekt sám, tedy v následujícím kroku je vyhodnocován výraz `< 3`. Ten je tvořen binární zprávou `<` s implicitním cílovým objektem. Virtuální stroj začne hledat slot `<` v aktivačním objektu, kde ovšem není. Avšak existuje zde rodičovský slot `:self*`. Hledání proto pokračuje v objektu `17`, na který slot ukazuje, a v dalších nadřazených objektech v hierarchii dědičnosti. Nakonec je nalezen požadovaný binární slot `<` v objektu `traits smallInt`. Metoda ve slotu je vyhodnocena (aktivována v kontextu objektu `17` s parametrem `3`) a vrací objekt `false`, což je současně návratová hodnota výrazu `< x`. Získáváme tím výraz `false ifTrue: self False: x`. Jednoduché podvýrazy `self` a `x` jsou vyhodnoceny na objekty `17` a `3` a vzniká výraz

```
false ifTrue: 17 False: 3
```

zasílající zprávu se selektorem `ifTrue:False:` objektu `false`. Přímo v objektu `false` je nalezen slot `ifTrue:False:` tvaru

```
ifTrue: b1 False: b2 = (b2 value)
```

Metoda ve slotu po aktivaci vrátí objekt `3`, což je i konečný výsledek aktivace metody `min:`. Aktivační objekt metody `min:` je zahozen a celý výraz

```
17 min: 3
```

vrací objekt `3`.

2.4.3 Aktivace bloku

Aktivace bloku je o něco složitější než aktivace metody, protože blok musí zachovávat vazbu na aktivační objekt, v kterém byl vytvořen a na který se můžou odkazovat zprávy uvnitř bloku. Pro ilustraci budeme uvažovat metodu

```
inc: x = ( [ |:y| y + x ] value: 3 )
```

aktivovanou ve výrazu

```
inc: 2
```

Označme si aktivační objekt této metody jako `acinc`. Při vyhodnocování výrazu v metodě dojde k vytvoření bloku

```
[ |:y| y + x ]
```

Při tvorbě bloku jsou vytvořeny dva objekty: *datový objekt bloku* a *metoda bloku*. Datový objekt bloku obsahuje, jak už bylo popsáno v oddílu 2.1.3, slot `parent*` inicializovaný na objekt `traits block` (obsahující metody sdíleného chování všech bloků), dále anonymní slot `<lexicalParent>` inicializovaný na lexikálně nadřazený aktivační objekt (v tomto případě objekt `acinc`), a slot se selektorem tvaru `value[:[With:[With:[...With:]]]]`. Selektor tohoto slotu je generován na základě počtu parametrických slotů bloku, v našem příkladu tedy bude mít tvar `value:`, protože blok obsahuje jeden parametr (`:y`). Do slotu `value:` je vložen odkaz na metodu bloku. Metoda bloku obsahuje anonymní slot `<parent>*`, sloty parametrů bloku a kód bloku. Dohromady tedy příkaz

```
[ |:y| y + x ]
```

vytvoří strukturu objektů

```
( |
        parent* = traits block.
        <lexicalParent> = acinc.
        value: = ( |:<parent>*. :y| y + x )
| )
```

Metoda bloku na rozdíl od běžné metody neosahuje implicitní slot `:self*`.

Při aktivaci bloku (zasláním zprávy `value: 3` datovému objektu bloku) dojde k vytvoření *kopie* metody bloku – vzniká *aktivační objekt metody bloku*. Slot `:parent*` je inicializován na objekt uložený slotu `<lexicalParent>`, tedy `acinc` (aktivační kontext, v

kterém blok vznikl). Do parametrických slotů aktivačního objektu metody bloku jsou vloženy odkazy na parametry zprávy `value:`; v našem případě se do slotu `:y` vloží reference na `3`. Celý aktivační objekt metody bloku vypadá takto:

```
( |:<parent>* = acinc. :y = 3| y + x)
```

Poté je standardním způsobem vyhodnocen kód aktivačního objektu metody bloku. To znamená, že se začne vyhodnocovat výraz `y + x`. Slot podvýrazu `y` je nalezen přímo v aktivačním objektu a nese objekt `3`. Slot podvýrazu `x` však v aktivačním objektu není. Hledání proto pokračuje v rodičovském slotu `:<parent>*`, to znamená v objektu `acinc`. V něm je nalezen slot `:x`, který byl při aktivaci zprávy `inc: 2` inicializovaný na objekt `2`. Dostáváme výraz

```
3 + 2
```

který je vyhodnocen na objekt `5`, výsledek aktivace bloku `[ |:y| y + x ] value: 3` v kontextu aktivačního objektu metody `inc: 2`.

V testované implementaci SELFu (verze 4.1) není možné, aby blok opustil rozsah platnosti lexikálně nadřazeného aktivačního objektu, avšak autoři předpokládají, že toto omezení bude v dalších verzích sníženo a u bloků bez úzké vazby na nadřazený aktivační kontext zcela odstraněno.

2.4.4 Přeposílání zpráv

Mechanismus přeposílání zpráv umožňuje explicitní zaslání zprávy rodičovským objektům. Přeposlání se uplatní především při volaní metod v nadřazených objektech, které byly v aktuálním objektu předefinovány. Přeposílání zpráv v SELFu odpovídá odpovídá konstrukci `super` v Smalltalku.

Nesměrované přeposlání zprávy má tvar

```
resend.factorial.
resend.+ 7.
resend.min: 3 Max: 15
```

kde za klíčovým slovem `resend` následuje tečka a zasílaná zpráva.

Směrované přeposlání zprávy umožňuje poslat zprávu konkrétnímu rodičovskému objektu:

```
intParent.factorial.
anotherParent.+ 7.
parent.min: 3
```

Přeposlání zprávy je virtuálním strojem realizováno tak, že hledání slotu se jménem selektoru zprávy přeskakuje objekt samotný a začíná až ve všech rodičovských objektech (u nesměrovaného přeposlání), resp. v konkrétním rodičovském objektu (u směrovaného přeposlání).

Přeposílat lze pouze zprávy s implicitním cílovým objektem.

3 Self World

Světem Selfu nazývají autoři Selfu rozsáhlou množinu objektů tvořících nedílnou součást celého systému Selfu. Kolekce předdefinovaných objektů zahrnuje jednoduché datové struktury, prostředky uživatelského prostředí, ladící nástroje i celé aplikace. Svět Selfu však nezahrnuje pouze tyto objekty, ale zároveň definuje smluvené principy organizace objektů a jejich používání. V následujících oddílech budou popsány význačné objekty systému a konvence požívané při organizaci světa Selfu.

3.1 Objekt lobby

Jak už víme, v Selfu existují pouze objekty a sloty, což jsou jednoduše reference na objekty. Každý objekt je dostupný pouze prostřednictvím slotů, které na něj ukazují. Objekty, na které se nelze odnikud odkázat, se považují za zrušené a podléhají automatickému garbage collectingu. Self tedy používá techniku perzistence na základě dosažitelnosti. Perzistentním kořenem a vstupním bodem do celého světa Selfu je objekt `lobby`.

Objekt `lobby` obsahuje několik slotů, odkazujících na další významné objekty. Těmito sloty jsou `lobby`, `globals`, `defaultBehavior`, `traits`, `mixins`, a `shell`.

Obrázek 2: objekt lobby

3.1.1 Slot lobby

Slot `lobby` obsahuje referenci zpět na objekt `lobby`. Zajišťuje tak pojmenování objektu `lobby` a zpřístupňuje jej všem objektům dědícím z objektu `lobby`.

3.1.2 Objekt globals

Protože v Selfu neexistují třídy, musí existovat mechanismus, jak vytvářet instance nových objektů s určitým sdíleným chováním, aniž by bylo nutné pro každý objekt znovu definovat všechny metody a sloty. Self pro účely instanciace nových objektů jednoho datového typu používá princip *klonování prototypů*. Ve světě Selfu proto existuje od každé „třídy“ objektů jeden objekt jako prototyp a nové objekty stejné „třídy“ se získávají jeho kopírováním.

Slot `globals` objektu `lobby` ukazuje na objekt `globals`, který slouží jako *sklad prototypů*. Obsahuje obrovské množství slotů ukazujících na prototypy všech objektů systému.

Jsou zde odkazy na všechny aplikace, objekty datových struktur, objekty interních prostředků SELFu a mnoho dalších.

Slot `globals` je v objektu `lobby` definován jako rodičovský. Protože (až na speciální výjimky) mají všechny objekty někde v hierarchii dědičnosti objekt `lobby`, umožňuje rodičovský slot `globals` snadné vytváření nových objektů. Například pro instanciaci objektu datového typu `list` (obousměrně vázaného seznamu) lze kdekoliv vyvolat zprávu `list copy`, která aktivuje slot `list` v objektu `globals`, ukazující na prototyp objektu seznamu. Tomu je zaslána zpráva `copy`, která vytvoří jeho mělkou kopii a vrátí její referenci.

3.1.3 Objekt defaultBehavior

Objekt `defaultBehavior` definuje chování, které by mělo být společné pro (téměř) všechny objekty vyskytující se ve světě SELFu. Obsahuje metody pro porovnávání identity, základní konverze, správu chyb, klonovací metodu `clone`, odkazy na stdin, stdout, stderr, metody pro tisk objektu a další.

Slot `defaultBehavior` je v objektu `lobby` opět definován jako rodičovský. Tím je zajištěno, že objekt `lobby` a všechny od něj dědící objekty sdílí toto společné chování.

3.1.4 Objekt traits

Podobně jako objekt `globals` udržuje množinu všech prototypů objektů v systému, objekt `traits` obsahuje prototypy sdíleného chování, tj. ekvivalenty „tříd“, známých z jiných objektově orientovaných programovacích jazyků. V SELFu existuje konvence, že každý datový typ je definován dvěma objekty: prototypem objektu uloženém v `globals` a objektem definujícím chování uloženým v objektu `traits`. Prototyp objektu pak obsahuje rodičovský slot (podle konvence obvykle nazvaný `parent*`), který ukazuje na patřičný traits objekt daného datového typu.

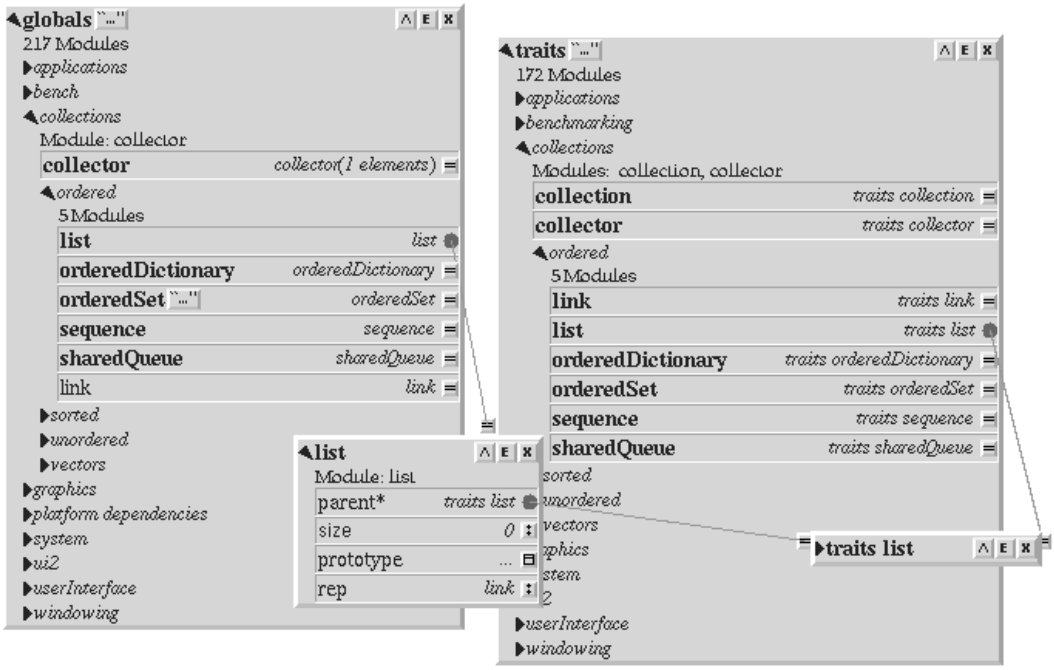
Uvažujme opět datový typ obousměrně vázaného seznamu. Prototyp `list` v objektu `globals` obsahuje slot `parent*` ukazující na objekt `traits list` a pouze tři další sloty reprezentující vnitřní data seznamu. Veškeré metody pro práci se seznamem jsou uloženy v `traits list`, odkud je dědí rodičovským slotem všechny kopie prototypu `list`.

3.1.5 Objekt mixins

Objekt `mixins` má podobný význam jako objekt `traits`; obsahuje objekty definující sdílené chování. Na rozdíl od traits objektů však „mixiny“ nedefinují chování celého datového typu, ale pouze určité speciální schopnosti, které můžou být „přimixovány“ k objektům za účelem specifických rozšíření.

Příkladem může být například mixin `mixins ordered`, definující operátory uspořádání (`<=`, `>=`, `>` apod.) Pokud některý objekt dědí od tohoto mixinu, dává tím najevo, že jej lze porovnávat s jinými objekty ve smyslu uspořádání.

Ekvivalentem mixinů SELFu jsou například mixins v jazyce Ruby nebo (do jisté míry) interfaces v jazyce C#.



Obrázek 3: objekty globals, traits a prototyp list

3.1.6 Objekt shell

Posledním slotem v `lobby` je slot ukazující na objekt `shell`. V kontextu tohoto objektu obvykle začíná interakce se systémem Self, je to také první objekt, který je otevřen při vstupu do grafického uživatelského rozhraní.

3.2 Traits clonable a traits oddball

Drtivá většina objektů nepřímo dědí z jednoho z objektů `traits clonable` nebo `traits oddball`, případně z jejich variant s přimixovaným uspořádáním `traits orderedClonable` a `traits orderedOddball`.

Objekty dědící od `traits oddball` resp. `traits orderedOddball` jsou tzv. *unikátní* objekty (často také označované jako singletony); v celém systému existuje jejich jediná instance. Zpráva `copy` zaslaná unikátnímu objektu nevytváří kopii, ale vrací opět referenci na stejný objekt. Příkladem unikátních objektů jsou například objekty `true`, `false` nebo čísla (existuje jediná instance např. objektu `13`, a všechny odkazy na číslo 13 ukazují na tento objekt).

Objekty dědící od `traits clonable` nebo `traits orderedClonable` nejsou unikátní a lze vytvářet neomezeně mnoho jejich mělkých kopií. Tento charakter má většina „normálních“ objektů.

Poznámka: při podrobnějším průzkumu např. objektu `traits clonable` lze zjistit, že obsahuje pouze čtyři sloty: `parent*` ukazující na `lobby`, `cloning*` ukazující na `mixins clonable`, `comparing*` ukazující na `mixins identity` a `ordering*` ukazující na `mixins unordered`. Objekt `traits clonable` tedy sám žádné chování neimplementuje, ale slu-

čuje dohromady požadované vlastnosti definované v několika mixinech. Toto je obvyklá technika používaná v řadě standardních objektů SELFu, zajišťující vysokou flexibilitu a znovupoužitelnost kódu. Důležitá je i přítomnost slotu `parent*` ukazujícího na `lobby`. Děděním z objektu `traits clonable` tedy získáváme i nepřímou dědičnost z `lobby`; právě tímto dědením je zajištěn dříve zmiňovaný fakt, že většina objektů SELFu dědí z `lobby`.

3.3 Objekty true a false, podmíněné příkazy

Logické hodnoty *true* a *false* jsou reprezentovány unikátními objekty `true` a `false`. Tyto objekty dědí z `traits boolean` a poskytují sloty s operátory `&&` (logický součin), `||` (logický součet) a `^^` (nonekvivalence).

Navíc obsahují metody `ifTrue:`, `ifFalse:` a `ifTrue:False:`, kterými se v SELFu realizuje větvení kódu. Parametry těchto metod jsou objekty nebo bloky, které se vyhodnotí resp. přeskočí, podle toho, zda je zpráva zaslána objektu `true` nebo `false`.

3.4 Bloky, podmíněné cykly

Jak již bylo zmíněno v oddílu 2.4.3, všechny bloky dědí od objektu `traits block`. Tento objekt, kromě jiného, implementuje i zprávy `whileTrue:`, `whileFalse:`, `untilTrue:`, `untilFalse:`, a `loop`, kterými se v SELFu realizují cykly.

Například zpráva `whileTrue:` reprezentuje klasický while-cyklus. Uvažujme metodu

```
(
        | x | x: 7.
        [ x > 0 ] whileTrue: [ x: x-1. 'Hello World' printLine. ]
)
```

Tato metoda vytiskne sedmkrát na standardní výstup text `Hello World`.

3.5 Čísla, počítané cykly

Čísla v SELFu jsou unikátní objekty nepřímo dědící z objektu `traits number`. Základními typy jsou čísla dědící `traits float`, `traits smallInt` a `traits bigInt`. Tyto objekty poskytují metody a operátory pro obvyklé operace s čísly. Navíc obsahují i metody pro počítané cykly (for-cykly). Jsou to například `to:Do:`, `to:By:Do:`, `upTo:Do:` a další.

V systému (přesněji řečeno v `globals`) existují navíc tři speciální objekty, `infinity`, `minSmallInt` a `maxSmallInt`. První reprezentuje nekonečno, druhé dvě udávají minimum a maximum datového typu `smallInt`.

Příklad počítaného cyklu:

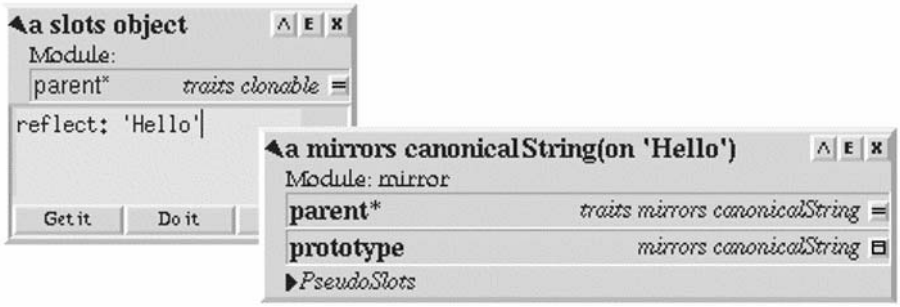
```
(
        1 to: 7 Do: [ 'HelloWorld' printLine. ]
)
```

V tomto příkladu je zaslána číselnému objektu `1` zpráva `to:Do:` s parametry `7` a `[ 'HelloWorld' printLine. ]`.

3.6 Reflexe

Reflexe je v Selfu podporována kolekcemi dědícími od `traits mirrors`. Zasláním zprávy `reflect: x` libovolnému objektu dědícímu od `defaultBehavior` lze získat tzv. *zrcadlový objekt* (mirror) objektu `x`. Zrcadlový objekt je vlastně asociativní pole mapující řetězce odpovídající názvům slotů v objektu `x` na zrcadlové objekty od objektů uložených v těchto slotech. Prohlížením a manipulací s touto kolekcí lze prohlížet a manipulovat se sloty v objektu `x`. Přes zrcadlový objekt lze do objektu přidávat a odebírat sloty, popř. měnit jejich obsah.

Podpora zrcadlových objektů a jejich implementace prostřednictvím sady primitiv jsou součástí virtuálního stroje. Virtuální stroj poskytuje zrcadlové objekty od těchto kategorií objektů v systému: čísla typu smallInt a float, řetězce, vektory, zrcadlové objekty (lze provádět i reflexi zrcadlových objektů), bloky, obyčejné metody a metody bloků, aktivační objekty metod a bloků, procesy, profilery, přiřazovací primitivum a „obyčejné“ objekty.



Obrázek 4: zrcadlový objekt objektu 'Hello'

4 Srovnání Selfu s běžnými objektově-orientovanými jazyky

Tento oddíl obsahuje souhrn nejobvyklejších technik, známých z jiných objektově-orientovaných programovacích jazyků, a způsoby jejich realizace v jazyce Self.

4.1 Objekty a třídy, instanciace

Self, na rozdíl od většiny běžných objektově-orientovaných jazyků, nezná pojem třídy. Sdílené chování objektů stejného datového typu je realizováno dědičností od traits objektů. Nové objekty se obvykle vytváří klonováním prototypů uložených v objektu `globals`. Protože rozhraní objektu není vázáno na žádnou třídu, strukturu objektů (sloty a jejich obsah) lze měnit (přes zrcadlové objekty) po celou dobu existence objektu.

4.2 Metody, členské proměnné

Self nerozlišuje metody a proměnné objektu. Objekt obsahuje pouze množinu pojmenovaných slotů a chování při aktivaci slotu (zaslání zprávy) závisí na objektu, který je na

slot navázán. Self tedy umožňuje dokonalé skrývání implementace chování.

4.3 Typová kontrola

Protože v Selfu neexistuje pojem datového typu, typová kontrola při překladu, známá z jazyků jako C++, Java nebo C# zde neexistuje. Překladač akceptuje jakýkoliv syntakticky správný kód. V době vykonávání programu pak může dojít jen k jedinému typu chyby, kdy virtuální stroj při zaslání zprávy nenajde slot s odpovídajícím názvem, nebo naopak najde více než jeden. Na obrázku 5 je ukázka reakce virtuálního stroje na „typovou“ chybu, kdy je k číslu `17` přičítán řetězec `'Hello'`.

Obrázek 5: lookup chyba při zaslání nesmyslné zprávy

4.4 Dědičnost, vícenásobná dědičnost

Self podporuje dědičnost prostřednictvím rodičovských slotů. Dědičnost v Selfu není ničím omezena, závisí pouze na výskytu rodičovských slotů v objektu. Pokud objekt obsahuje jediný rodičovský slot, typicky nazývaný `parent*`, pak dědí od jednoho nadřazeného objektu (jednoduchá dědičnost). Obsahuje-li objekt více rodičovských slotů, jedná se o vícenásobnou dědičnost. Vztah dědičnosti mezi objekty může tvořit libovolný orientovaný graf, včetně smyček a cyklů. Pokud je rodičovský slot vytvořen pro čtení i zápis, pak takový objekt používá dynamickou dědičnost (je možné měnit dynamicky v průběhu existence objektu jeho rodiče).

4.5 Redefinice, polymorfismus

Z principu vyhledávacho algoritmu při zaslání zprávy plyne, že se zastaví na prvním slotu s odpovídajícím názvem. Redefinice děděného slotu se proto v Selfu provede jednoduše vytvořením tohoto slotu v objektu, který jej má předefinovat. Protože Self nerozlišuje mezi sloty ukazujícími na datové objekty a ukazujícími na metody, je dokonce možné původně datový slot redefinovat na slot metody a naopak, bez jakékoliv změny rozhraní objektu.

Z principu vyhodnocování výrazů současně vyplývá, že všechny zprávy mají automaticky polymorfní charakter.

4.6 Abstraktní bázové třídy

Jak již bylo řečeno, ekvivalentem tříd v Selfu jsou traits objekty. Ty mají samy o sobě charakter abstraktních tříd, protože obsahují pouze sloty definující sdílené abstraktní rozhraní a chování. Self sice nebrání vytváření jejich klonů, avšak tyto klony obvykle nelze plnohodnotně používat. (Typickým příkladem je třeba rozsáhlá hierarchie traits objektů pro kolekce.) Za abstraktní bázový objekt lze tedy v Selfu považovat jakýkoliv objekt, který definuje určité rozhraní a popř. i chování, ale nemá význam jej používat jinak než formou dědičnosti z jiných objektů.

5 Implementace Selfu

Celý systém Self je volně dostupný (včetně zdrojových kódů) na webových stránkách firmy Sun Microsystems (`http://research.sun.com/self`). Oficiální distribuce vyžaduje operační systém Mac OS X nebo pracovní stanici SPARC s operačním systémem Solaris. Poslední oficiální verze je Self 4.2. Součástí balíku je virtuální stroj a rozsáhlá sada knihoven tvořící svět Selfu. Pro komfortní práci se systémem Self existuje grafické uživatelské rozhraní *Morphic*, z kterého pochází všechny obrázky v tomto dokumentu.

Kromě standardní distribuce pro Mac OS X a Solaris existuje i port virtuálního stroje pro procesory Intel x86, vytvořený pro operační systém Linux, popř. Windows (s použitím knihovny cygwin). Jeho hlavními autory jsou Harald Gliebe a Gordon Cichon a je k dispozici na `http://www.gliebe.de/self`. Port je ve vývojovém stadiu, nicméně virtuální stroj je poměrně stabilní a funkční. Hlavním nedostatkem portu je v době psaní tohoto dokumentu rychlost, protože současná verze disponuje pouze neinlinujícím kompilátorem (NIC), optimalizující SIC kompilátor (simple inlining compiler) je v experimentálním stadiu. V důsledku toho je grafické prostředí Selfu běžící nad tímto portem velmi pomalé a náročné na výkon procesoru (např. ve srovnání se souběžně testovanou komerční implementací Smalltalku VisualWorks).

Reference

[1] *The* Self *4.1 Programmer's Reference Manual*, Sun Microsystems, Inc., 1995-2000.

[2] David Ungar, *How To Program in* Self *4.1*, Sun Microsystems, Inc., 2000.

[3] John Maloney, *Morphic: The* Self *User Interface Framework*, Sun Microsystems, Inc., 2000.