Separační logika

Separační logika

Separační logika (**separation logic**) je variantou Hoareovy logiky, která umožňuje snadněji než standardní Hoareova logika verifikovat programy, které pracují s ukazateli do paměti a s datovými strukturami.

- Použití standardní Hoareovy logiky pro takový druh programů je sice principiálně možné, ale i pro poměrně malé programy jsou pak důkazy značně komplikované.
- U programů, které pracují s ukazateli do paměti, není možné používat standardní pravidlo Hoareovy logiky pro přiřazení pro operace, které zapisují na adresu danou ukazatelem, nebo které z takové adresy čtou.
- Je třeba udržovat informace o tom, kam mohou jednotlivé ukazatele do paměti ukazovat a zda ukazují či neukazují na stejná místa v paměti.

Separační logika

Hlavním problémem je tzv. **aliasing**:

- Řekněme, že p a q jsou dvě proměnné obsahující ukazatele do paměti.
- Pokud oba ukazatele ukazují na stejné místo v paměti (což se označuje jako aliasing), zápis hodnoty, na kterou ukazuje ukazatel p, změní i hodnotu, na kterou ukazuje ukazatel q.
- Pokud q ukazuje na nějakou jinou adresu, hodnota na kterou ukazuje q, se při daném zápisu do paměti nezmění.
- Obecně není vždy jednoduché určit, který z těchto případů nastává.

V běžné Hoareově logice by bylo třeba tyto informace o možném aliasingu udržovat explicitně ve formulích, které by tak musely obsahovat informace i o všech možných částech paměti, které s danou částí kódu vůbec nemusí souviset.

Separační logika

Separační logika toto řeší tak, že:

- Formule se nevyjadřují o globálním stavu paměti, ale vždy jen o části paměti, se kterou daná část kódu pracuje.
- Rozšiřuje běžnou predikátovou logiku o nové druhy logických operátorů, jako jsou **separační konjunkce** a **separační implikace**, které umožňují ve struktuře formule **implicitně** zachytit informaci o tom, že určité části paměti, na které ukazují dané ukazatele, se spolu nepřekrývají a nedochází tam k aliasingu.
- Na rozdíl od standardní Hoareovy logiky umožňuje více modulární přístup, kdy formule v Hoareových trojicích, musí explicitně zmiňovat tu část paměti, ze kterou daný kód pracuje, ale implicitně je dáno, že zbylá část paměti se provedením daného kódu nemění, a kód k ní nikdy nepřistupuje.

Sémantika programů pracujících s pamětí

Uvažujme programovací jazyk, kde:

- Var — množina proměnných
- $Value$ — množina hodnot, které mohou být přiřazeny do proměnné nebo uloženy v jedné buňce paměti
- Loc — množina adres paměti

Předpokládáme, že **paměť** je tvořena buňkami:

- buňky jsou adresovány pomocí adres z množiny Loc
- každá buňka paměti může obsahovat hodnotu z množiny $Value$
- adresy z množiny Loc jsou speciálním případem hodnot z množiny $Value$

Sémantika programů pracujících s pamětí

Pro konkrétnost si například můžeme představovat, že:

- Loc je množina přirozených čísel
- $Value$ zahrnuje i další druhy hodnot, např. celá čísla, booleovské hodnoty, apod.

Můžeme také rozlišovat dvě následující varianty:

- Typ adres Loc je samostatný typ odlišný od běžných přirozených či celých čísel, tj. například

  $$Value = Loc \cup \mathsf{int} \cup \mathsf{bool} \cup \cdots$$

  (přičemž množina Loc je disjunktní s množinami int, bool, atd.)

- Adresy i čísla jsou jedno a totéž. Tj. například

  $$Value = Loc = \mathbb{N}$$

Sémantika programů pracujících s pamětí

Stav běžícího programu je dán jako dvojice

$$\langle \sigma, m \rangle$$

kde:

- σ — funkce typu $Var \rightharpoonup_{fin} Value$
  určuje **přiřazení hodnot proměnným**
- m — funkce typu $Loc \rightharpoonup_{fin} Value$
  určuje **obsah paměti**, tj. přiřazení hodnot adresám paměti

Poznámka: Zápis $A \rightharpoonup_{fin} B$ zde označuje parciální funkce z A do B, ve kterých je hodnota dané funkce definována jen pro nějakou **konečnou** podmnožinu množiny A (přičemž množina A může být nekonečná).

Sémantika programů pracujících s pamětí

Příklad: Stav daný dvojicí $\langle \sigma, m \rangle$, kde

- Přiřazení hodnot proměnným:

$$\sigma = [X \mapsto 42;\ Y \mapsto 0;\ W \mapsto -5]$$

Tedy:

  - proměnná X obsahuje hodnotu 42
  - proměnná Y obsahuje hodnotu 0
  - proměnná W obsahuje hodnotu -5

- Obsah paměti:

$$m = [42 \mapsto -3;\ 43 \mapsto 75;\ 75 \mapsto 17;\ 76 \mapsto 0]$$

Tedy:

  - buňka s adresou 42 obsahuje hodnotu -3
  - buňka s adresou 43 obsahuje hodnotu 75
  - buňka s adresou 75 obsahuje hodnotu 17
  - buňka s adresou 76 obsahuje hodnotu 0

Sémantika programů pracujících s pamětí

- Proměnné jsou oddělené od paměti.

  Mohou ale obsahovat jako hodnoty adresy buněk paměti (tj. hodnoty typu Loc).

  Proměnná obsahující hodnotu typu Loc představuje **ukazatel** (**pointer**) do paměti.

- Také buňky paměti mohou (kromě jiných dalších hodnot) obsahovat ukazatele do paměti.

- Obecně je možné s ukazateli do paměti provádět **pointerovou aritmetiku**, převádět je z čísel a na čísla, apod.

Sémantika programů pracujících s pamětí

Poznámka: Ve variantách separační logiky, které se používají pro reprezentaci nízkoúrovňových jazyků, jako např. strojového kódu nebo jazyka C, se používá reprezentace paměti, která ještě věrněji odráží organizaci paměti skutečných počítačů:

- Buňky odpovídají bytům paměti.
- Adresy jsou 32-bitová nebo 64-bitová čísla.
- Jedno 32-bitové číslo zabírá 4 po sobě jdoucí buňky paměti, apod.

Zde ale pro jednoduchost budeme uvažovat idealizovaný počítač, kde adresy, obsahy buněk i hodnoty v proměnných mohou být libovolně velká čísla.

Sémantika programů pracujících s pamětí

S hodnotami σ a m reprezentujícími stav běžícího programu $\langle \sigma, m \rangle$ budeme provádět následující operace:

- $\sigma(X)$ — kde $X \in Var$

  reprezentuje hodnotu proměnné X v daném stavu

- $\sigma[X \mapsto v]$ — kde $X \in Var$ a $v \in Value$

  reprezentuje nové přiřazení hodnot proměnným, které proměnné X přiřazuje hodnotu v a všem ostatním proměnným stejné hodnoty jako přiřazení σ

- $m(p)$ — kde $p \in Loc$

  reprezentuje hodnotu uloženou na adrese p

- $m[p \mapsto v]$ — kde $p \in Loc$ a $v \in Value$

  reprezentuje nový obsah paměti, kde buňka na adrese p obsahuje hodnotu v a všechny ostatní buňky obsahují stejné hodnoty jako při obsahu paměti m

Sémantika programů pracujících s pamětí

- Zápisem $dom(f)$, kde $f : A \rightharpoonup_{fin} B$, budeme označovat **definiční obor** (**domain**) parciální funkce f, tj. podmnožinu těch prvků množiny A, pro které je hodnota funkce definovaná.

  Například pro následující obsah paměti

  $$m = [42 \mapsto -3;\ 43 \mapsto 75;\ 75 \mapsto 17;\ 76 \mapsto 0]$$

  bude

  $$dom(m) = \{42, 43, 75, 76\}$$

Sémantika programů pracujících s pamětí

Uvažujme programovací jazyk s podobnou syntaxí a sémantikou, jako byl dříve popsaný jazyk Imp, který ale bude rozšířen o následující čtyři příkazy pro práci s pamětí:

- **Load** — přečte obsah buňky, jejíž adresa je určená hodnotou výrazu a, a tuto hodnotu uloží do proměnné X:

$$X := [\, a \,]$$

- **Store** — hodnotu výrazu a_2 zapíše do buňky, jejíž adresa je určená hodnotou výrazu a_1:

$$[\, a_1 \,] := a_2$$

Sémantika programů pracujících s pamětí

- **Alokace paměti** — alokuje nové dosud nepoužité buňky na adresách p, $p{+}1$, …, $p{+}k{-}1$, do kterých uloží hodnoty výrazů a_0, a_1, …, a_{k-1}.

  Adresu p udávající začátek alokovaného bloku buněk přiřadí do proměnné X:

  $$X := \mathrm{cons}\,(a_0, a_1, \ldots, a_{k-1})$$

  Poznámka: Konkrétní adresa p je určena nedeterministicky.

- **Uvolnění buňky paměti** — uvolní buňku paměti, jejíž adresa je dána hodnotou výrazu a:

  $$\mathrm{dispose}\,(a)$$

Poznámky:

- Kromě výše uvedených instrukcí můžeme obecně uvažovat i další druhy instrukcí, např. alokaci pole (tj. souvislého úseku buněk, jehož délka může být dána hodnotou výrazu), obecně práci s poli, práci se záznamy (struct, record), apod.

- Uvažujeme zde případ jazyka s explicitní alokací a dealokací paměti.

  Existují i varianty separační logiky zaměřené na jazyky používající **garbage collector**.

Sémantika programů pracujících s pamětí

Sémantiku daného jazyka můžeme formálně popsat pomocí **big-step** nebo **small-step** sémantiky podobně, jako tomu bylo u dříve popsané verze jazyka Imp.

Například při použití big-step sémantiky budeme mít sadu pravidel tvaru:

$$\langle \sigma, m \rangle \ \rightrightarrows \ c \ \Rightarrow \ \langle \sigma', m' \rangle$$

kde $\langle \sigma, m \rangle$ reprezentuje stav před provedením příkazu c
a $\langle \sigma', m' \rangle$ stav po provedení tohoto příkazu.

Většina pravidel bude hodně podobných, jako v případě jazyka Imp, například:

$$\frac{\sigma \vdash a \Rightarrow v}{\langle \sigma, m\rangle \ \Rightarrow\!\!\langle\, x \;:=\; a \,\rangle\!\!\Rightarrow \ \langle \sigma', m\rangle} \quad \text{kde } \sigma' = \sigma[x \mapsto v]$$

$$\frac{\langle \sigma, m\rangle \ \Rightarrow\!\!\langle\, c_1 \,\rangle\!\!\Rightarrow \ \langle \sigma', m'\rangle \qquad \langle \sigma', m'\rangle \ \Rightarrow\!\!\langle\, c_2 \,\rangle\!\!\Rightarrow \ \langle \sigma'', m''\rangle}{\langle \sigma, m\rangle \ \Rightarrow\!\!\langle\, c_1 \;;\; c_2 \,\rangle\!\!\Rightarrow \ \langle \sigma'', m''\rangle}$$

$$\frac{\sigma \vdash b \Rightarrow \text{true} \qquad \langle \sigma, m\rangle \ \Rightarrow\!\!\langle\, c_1 \,\rangle\!\!\Rightarrow \ \langle \sigma', m'\rangle}{\langle \sigma, m\rangle \ \Rightarrow\!\!\langle\, \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2 \,\rangle\!\!\Rightarrow \ \langle \sigma', m'\rangle}$$

Sémantika programů pracujících s pamětí

- Pravidlo pro instrukci **load**:

$$\frac{\sigma \vdash a \Rightarrow v_1 \qquad m(v_1) = v_2}{\langle \sigma, m \rangle \; =\!\!\langle \; x := [\, a \,] \; \rangle\!\!\Rightarrow \; \langle \sigma', m \rangle} \quad \text{kde } \sigma' = \sigma[x \mapsto v_2]$$

- Pravidlo pro instrukci **store**:

$$\frac{\sigma \vdash a_1 \Rightarrow v_1 \quad v_1 \in dom(m) \quad \sigma \vdash a_2 \Rightarrow v_2}{\langle \sigma, m \rangle \; =\!\!\langle \; [\, a_1 \,] := a_2 \; \rangle\!\!\Rightarrow \; \langle \sigma, m' \rangle} \quad \text{kde } m' = m[v_1 \mapsto v_2]$$

Assertions

Assertions nejsou v separační logice vyhodnocovány vůči globálním stavům $\langle \sigma, m \rangle$, ale vůči dvojicím tvaru

$$\langle \sigma, h \rangle$$

kde:

- σ — je přiřazení hodnot proměnným (stejně jako v globálním stavu)

  Poznámka: Toto přiřazení hodnot proměnným se často označuje pojmem **store**.

- h — je tzv. **heap** (někdy se též používá pojem **heaplet**)

  Je to funkce typu $Loc \rightharpoonup_{fin} Value$ (stejně jako obsah paměti m).

  Na rozdíl od globálního obsahu paměti m však heap h představuje obecně jen nějakou část obsahu paměti, ne nutně celou paměť.

Heaps

Příklad: Řekněme, že obsah paměti m je

$$m = [36 \mapsto 5;\ 37 \mapsto 0;\ 42 \mapsto -3;\ 43 \mapsto 75;\ 74 \mapsto 17;\ 75 \mapsto 36]$$

Příklady toho, jak mohou vypadat některé heaps odpovídající danému obsahu paměti m:

- $h_1 = [42 \mapsto -3;\ 43 \mapsto 75]$
- $h_2 = [\,]$
- $h_3 = [36 \mapsto 5;\ 37 \mapsto 0;\ 42 \mapsto -3;\ 43 \mapsto 75;\ 74 \mapsto 17;\ 75 \mapsto 36]$
- $h_4 = [37 \mapsto 0;\ 74 \mapsto 17]$

Assertions

Označme:

- $Stores$ — množinu všech možných stores (tj. všech možných přiřazení hodnot proměnným

  tedy množinu všech parciálních funkcí typu $Var \rightharpoonup_{fin} Value$

- $Heaps$ — množinu všech možných heaps

  tedy množinu všech parciálních funkcí typu $Loc \rightharpoonup_{fin} Value$

Formálně jsou pak **assertions** definovány jako predikáty typu

$$Stores \to Heaps \to \textbf{Prop}$$

To, že assertion P platí pro daný store σ a heap h, budeme označovat zápisem

$$\sigma, h \models P$$

Heaps

Pro definici toho, kdy platí $\sigma, h \models P$, budeme potřebovat následující parciální operaci $\oplus$ na prvcích množiny $Heaps$:

- Předpokládejme, že $h_1, h_2 \in Heaps$.
- Pokud $dom(h_1) \cap dom(h_2) = \emptyset$, pak je definován heap $h = h_1 \oplus h_2$, kde

$$h(p) = \begin{cases} h_1(p) & \text{pokud } p \in dom(h_1) \\ h_2(p) & \text{pokud } p \in dom(h_2) \end{cases}$$

  (Hodnota $h(p)$ není definována pro taková p, kde $p \notin dom(h_1) \cup dom(h_2)$.)

- Pokud $dom(h_1) \cap dom(h_2) \neq \emptyset$, pak hodnota $h_1 \oplus h_2$ není definována.

Heaps

Příklad: Řekněme, že

- $h_1 = [42 \mapsto -3;\ 43 \mapsto 75]$
- $h_2 = [26 \mapsto 5;\ 54 \mapsto 42;\ 56 \mapsto 0]$
- $h_3 = [43 \mapsto 75;\ 44 \mapsto 45;\ 45 \mapsto 17]$

Pak:

$$h_1 \oplus h_2 = [26 \mapsto 5;\ 42 \mapsto -3;\ 43 \mapsto 75;\ 54 \mapsto 42;\ 56 \mapsto 0]$$

Hodnota $h_1 \oplus h_3$ není definována.

Poznámka: Na vztah $h_1 \oplus h_2 = h$ se alternativně můžeme dívat tak, že daná trojice h_1, h_2, h je v ternární relaci $Join$ typu

$$Heaps \to Heaps \to Heaps \to \textbf{Prop}$$

tj. že platí $Join(h_1, h_2, h)$.

Assertions

- Formule vyjadřující assertions mohou obsahovat logické spojky a kvantifikátory stejně jako běžné formule predikátové logiky.
- Kromě toho mohou obsahovat některé další operátory specifické pro separační logiku, které umožňují se vyjadřovat o obsahu heapu.
- Formule, které neobsahují žádné z těchto speciálních operátorů, se označují jako **pure**.
- Pravdivostní hodnota pure formulí závisí pouze na storu σ, nikoli na heapu h.
- Pro pure formuli P tedy platí

$$\sigma, h \models P \quad \text{iff} \quad \sigma \models P$$

Assertions

Význam logických spojek ($\wedge$, $\vee$, $\rightarrow$, $\neg$, ...) a kvantifikátorů ($\forall$ a $\exists$) je stejný jako v predikátové logice, např.:

- $\sigma, h \models P \wedge Q$ iff $\sigma, h \models P$ a $\sigma, h \models Q$
- $\sigma, h \models P \vee Q$ iff $\sigma, h \models P$ nebo $\sigma, h \models Q$
- $\sigma, h \models \neg P$ iff neplatí $\sigma, h \models P$

Assertions

Ve formulích se mohou vyskytovat **termy** tvořené libovolnými logickými proměnnými, konstantami, funkčními symboly a také proměnnými z množiny Var.

- Tyto termy budeme označovat E, E_1, E_2, E', F, F_1, F', apod.
- Aritmetické a booleovské výrazy daného programovacího jazyka budeme považovat za speciální případ termů.
- Termy jsou **pure** v tom smyslu, že se neodkazují k obsahu paměti (resp. heapu).
- To, že se daný term E pro store σ vyhodnotí na hodnotu v budeme označovat zápisem

$$\sigma \models E \Rightarrow v$$

Assertions

Operátory specifické pro separační logiku:

- **emp** — formule, která platí právě tehdy, pokud je daný heap prázdný, tj.

$$\sigma, h \models \mathbf{emp} \quad \text{iff} \quad h = [\,]$$

- $E \mapsto F$ — formule, která platí právě tehdy, pokud je daný heap tvořen jedinou buňkou, jejíž adresa je dána hodnotou výrazu E a jejíž obsah je dán hodnotou výrazu F, tj.

$$\sigma, h \models E \mapsto F \quad \text{iff} \quad \sigma \models E \Rightarrow v_1,\ \sigma \models F \Rightarrow v_2 \text{ a } h = [\,v_1 \mapsto v_2\,]$$

Poznámka: Pro daný heap h tedy platí $dom(h) = v_1$.

Assertions

- $P * Q$ — tzv. **separační konjunkce**

  $\sigma, h \models P * Q$ platí právě tehdy, když je daný heap h možné rozdělit na dva heapy h_1 a h_2 takové, že:

  - $h = h_1 \oplus h_2$
  - $\sigma, h_1 \models P$
  - $\sigma, h_2 \models Q$

  Poznámka: Všimněte si, že aby mohlo platit $h = h_1 \oplus h_2$, musí být dané heapy **disjunktní**, tj. musí platit

$$dom(h_1) \cap dom(h_2) = \emptyset$$

Assertions

Příklad: Formule

$$(X \mapsto a) * (Y \mapsto b)$$

Tato formule vyjadřuje následující věci:

- proměnná X ukazuje obsahuje ukazatel, který ukazuje na buňku paměti obsahující hodnotu danou (logickou) proměnnou a
- proměnná Y ukazuje obsahuje ukazatel, který ukazuje na buňku paměti obsahující hodnotu danou (logickou) proměnnou b
- adresy těchto dvou buněk jsou různé
- v daném heapu se kromě těchto dvou buněk nenacházejí žádné další buňky

Assertions

- $P \mathrel{-\!\!*} Q$ — tzv. **separační implikace**

  $\sigma, h \models P \mathrel{-\!\!*} Q$ platí právě tehdy, když pro každý heap h_1, ve kterém platí P a který je možné přidat k danému heapu h, dostaneme tímto přidáním heap h_2, kde platí Q.

  Tj. pro každé h_1 takové, že:

  - $\sigma, h_1 \models P$
  - $dom(h_1) \cap dom(h) = \emptyset$

  platí:

  - $\sigma, h_2 \models Q$ kde $h_2 = h_1 \oplus h$

Assertions

Příklady některých ekvivalencí a pravidel, která platí pro formule separační logiky:

$$\begin{aligned} (P * Q) * R &\iff P * (Q * R) \\ P * Q &\iff Q * P \\ P * \mathbf{emp} &\iff P \end{aligned}$$

$$\frac{P \Rightarrow Q \qquad R \Rightarrow S}{P * R \Rightarrow Q * S}$$

$$\frac{P * R \Rightarrow S}{P \Rightarrow R \mathrel{-\!\!*} S} \qquad \frac{P \Rightarrow R \mathrel{-\!\!*} S \qquad Q \Rightarrow R}{P * Q \Rightarrow S}$$

$$(E_1 \mapsto F_1) * (E_2 \mapsto F_2) * \mathsf{True} \Rightarrow E_1 \neq E_2$$

Assertions

Separační konjunkce se v některých ohledech chová podobně jako „obyčejná“ konjunkce, ale v některých ohledech zase odlišně:

Například pro obyčejnou konjunkci platí:

$$P \wedge Q \Rightarrow P$$

$$P \wedge P \iff P$$

Analogická tvrzení pro separační konjunkci obecně **neplatí**:

$$P * Q \Rightarrow P$$

$$P * P \iff P$$

Speciálně například následující tvrzení je vždy nepravdivé:

$$(E \mapsto F) * (E \mapsto F)$$

Některé užitečné zkratky

Pro zkrácení zápisu formulí se hodí používat některé zkratky:

- $E \mapsto -$ – zkratka pro $\exists y.E \mapsto y$, kde $y \notin \mathit{free}(E)$
- $E \mapsto (F_0, F_1, \ldots, F_n)$ – zkratka pro
  $(E \mapsto F_0) * (E{+}1 \mapsto F_1) * \cdots * (E{+}n \mapsto F_n)$
- $E \hookrightarrow F$ – zkratka pro $(E \mapsto F) * \mathsf{True}$

  Tato formule vyjadřuje, že v daném heapu je na adrese určené hodnotou výrazu E uložená hodnota určená výrazem F, přičemž ale na rozdíl od formule $E \mapsto F$ mohou být v daném heapu přiřazeny hodnoty i jiným adresám než jen E.

Pure assertions

Připomeňme, že **pure** formule, jsou ty formule, které **neobsahují** žádný z operátorů vztahujících se k paměti, jako jsou:

- **emp**
- $E \mapsto F$
- $P * Q$
- $P \mathrel{-\!\!*} Q$

ani žádné další operátory, které jsou z nich odvozené.

Další užitečné zkratky:

- $[P]$, kde P je pure formule — zkratka pro formuli $P \wedge \mathbf{emp}$
- $[\,]$ — zkratka pro formuli **emp**

Hoareovy trojice

Hoareovy trojice tvaru

$$\{P\}\, c\, \{Q\}$$

mají v separační logice odlišnou sémantiku než v Hoareově logice.

V případě částečné korektnosti vyjadřuje daná trojice toto:

- Pro každé σ a h_1, kde platí $\sigma, h_1 \models P$, platí

  pro každé h_2 takové, že $dom(h_1) \cap dom(h_2) = \emptyset$, že pokud

  $$\langle \sigma, h_1 \oplus h_2 \rangle \;\preceq\; c \;\Longrightarrow\; \langle \sigma', h_1' \oplus h_2 \rangle$$

  pak

  $$\sigma', h_1' \models Q.$$

Pravidla separační logiky

V separační logice platí prakticky všechna pravidla jako v Hoareově logice, například:

$$\frac{P \Rightarrow P' \qquad \{P'\}\, c\, \{Q'\} \qquad Q' \Rightarrow Q}{\{P\}\, c\, \{Q\}}$$

$$\frac{\{P\}\, c_1\, \{Q\} \qquad \{Q\}\, c_2\, \{R\}}{\{P\}\, c_1 \,;\, c_2\, \{R\}}$$

$$\frac{\{P \wedge b\}\ c_1\ \{Q\} \qquad \{P \wedge \neg b\}\ c_2\ \{Q\}}{\{P\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{Q\}}$$

Pravidla separační logiky — přiřazení

Platí i Hoareovo pravidlo pro přiřazení:

$$\{P[a/X]\}\ X := a\ \{P\}$$

Je zde ale důležité, že výraz a je zde **pure**, tj. neobsahuje žádný přístup do paměti.

Alternativní možnost, jak vyjádřit pravidlo přiřazení v separační logice:

$$\{X = n \wedge \mathbf{emp}\}\ X := a\ \{X = (a[n/X]) \wedge \mathbf{emp}\}$$

Pokud místo $P \wedge \mathbf{emp}$ budeme psát $[P]$, bude toto pravidlo pro přiřazení vypadat takto (n je zde nová, dosud nepoužitá, proměnná):

$$\{[X = n]\}\ X := a\ \{[X = (a[n/X])]\}$$

Pravidla separační logiky — frame rule

Mimořádně důležitým pravidlem separační logiky je tzv. **frame rule**:

$$\frac{\{P\}\,c\,\{Q\}}{\{P * R\}\,c\,\{Q * R\}} \quad \text{kde } \mathit{Modifies}(c) \cap \mathit{free}(R) = \emptyset$$

kde:

- $\mathit{Modifies}(c)$ — množina všech proměnných programu, jejichž hodnota může být změněna provedením příkazu c
- $\mathit{free}(R)$ — množina všech volných proměnných ve formuli R

Pravidla separační logiky — práce s pamětí

Následující pravidla, označovaná jako **small axioms**, popisují činnost jednotlivých příkazů pracujících s pamětí:

- Load:

$$\{\,(E \mapsto n) \wedge (X = m)\,\}\ X := [E]\ \{\,(X = n) \wedge (E[m/X] \mapsto n)\}$$

- Store:

$$\{\,E \mapsto -\,\}\ [E] := F\ \{\,E \mapsto F\,\}$$

- Alokace paměti:

$$\{\,[X = m]\,\}\ X := \mathrm{cons}\,(E_1, \ldots, E_n)\ \{\,X \mapsto (E_1[m/X], \ldots, E_n[m/X])\,\}$$

- Uvolnění paměti (jedné buňky):

$$\{\,E \mapsto -\,\}\ \mathrm{dispose}\,(E)F\ \{\,\mathbf{emp}\,\}$$

Poznámka: Předpokládá se, že m a n jsou dvě nové dosud nepoužité proměnné.

Pravidla separační logiky — práce s paměti

V kombinaci s dalšími pravidly (hlavně např. s frame rule) je možné ze small axioms odvodit další pravidla, která mohou být při dokazování užitečnější než používat přímo small axioms:

Například:

- Store:

$$\{(E \mapsto -) * R\}\ [E] := F\ \{(E \mapsto F) * R\}$$

- Alokace paměti (předpokládáme, že $X \notin \mathit{free}(E_1, \ldots, E_n)$ a $X \notin \mathit{free}(R)$):

$$\{R\}\ X := \mathsf{cons}\,(E_1, \ldots, E_n)\ \{(X \mapsto (E_1, \ldots, E_n)) * R\}$$

Příklad odvození v separační logice

Příklad odvození ve formě anotovaného kódu:

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad \{\ \mathbf{emp}\ \} \\
X := \mathrm{cons}\,(a, a) \\
\qquad\qquad\qquad\qquad \{\ X \mapsto (a, a)\ \} \\
Y := \mathrm{cons}\,(b, b) \\
\qquad\qquad\qquad\qquad \{\ (X \mapsto (a, a)) * (Y \mapsto (b, b))\ \} \\
[X + 1] := Y - X \\
\qquad\qquad\qquad\qquad \{\ (X \mapsto (a, Y - X)) * (Y \mapsto (b, b))\ \} \\
[Y + 1] := X - Y \\
\qquad\qquad\qquad\qquad \{\ (X \mapsto (a, Y - X)) * (Y \mapsto (b, X - Y))\ \} \Rightarrow \\
\qquad\qquad\qquad\qquad \{\ \exists o.\, (X \mapsto (a, o)) * (X + o \mapsto (b, -o))\ \}
\end{array}
$$

Rekurzivně definované predikáty

V separační logice hrají důležitou roli **rekurzivně definivané predikáty**:

- Jedná se o speciální druh formulí, které jsou definovány rekurzivním způsobem.
- Slouží zejména k popisu různých datových struktur (např. různé druhy seznamů, stromů, apod.)
- Mohou popisovat, jak daná struktura v paměti odpovídá nějaké „idealizované“ matematické struktuře:

  např. jak seznam v paměti reprezentovaný pomocí buněk obsahujících hodnotu daného prvku a ukazatel na následující buňku odpovídá posloupnosti prvků chápané jako abstraktní matematický objekt

Rekurzivně definované predikáty

- Problematika rekurzivně definovaných predikátů je komplikovanější, protože obecně ne každý takto definovaný predikát představuje korektní definici — aby se jednalo o korektní definici, je třeba nejprve dokázat některé jeho další vlastnosti.

  Touto problematikou se zde v tomto stručném popisu separační logiky nebudeme podrobněji zabývat.

Rekurzivně definované predikáty

Jednoduchý příklad rekurzivně definovaného predikátu:

Následující predikát vyjadřuje to, že:

- daný heap obsahuje buňky jednosměrného seznamu (a nic dalšího)
- p představuje začátek tohoto seznamu
- ℓ je sekvence prvků tohoto seznamu

$$\mathsf{ListRepr}(p,\, \ell)$$

Tento predikát je definován následujícím způsobem:

$$\begin{aligned}\mathsf{ListRepr}(p,\, \ell) \quad \Longleftrightarrow \\ & [\, p = \mathsf{null} \;\wedge\; \ell = \mathsf{nil}\,] \;\vee \\ & (\exists x, \ell', q.\;\; [\, \ell = x :: \ell'\,] \;*\; p \mapsto (x, q) \;*\; \mathsf{ListRepr}(q, \ell'))\end{aligned}$$

Implementace separační logiky ze Software Foundations

Ve Volume 6 série Software Foundations:

Arthur Charguéraud — Separation Logic Foundations

je detailně popsána jedna konkrétní implementace separační logiky v Coqu.

Tato implementace:

- Nepoužívá imperativní jazyk, jaký byl popsán na předchozích slidech.
- Použitý jazyk je imperativní jazyk, který pracuje s pamětí, přičemž ale použitá syntaxe odpovídá funkcionálním jazykům (trochu se podobá jazyku OCaml).
- Jedná se tedy v principu o funkcinální jazyk, který ale může provádět vedlejší efekty na obsahu paměti.

Implementace separační logiky ze Software Foundations

Použití funkcinálního jazyka (s imperativními prvky) má oproti klasickému imperativnímu několik výhod.
Zejména se to týká toho, že celý systém je oproti implemetaci klasického imperativního jazyka v některých ohledech jednodušší:

- Stav systému je dán pouze obsahem paměti — součástí globálního stavu není přiřazení hodnot proměnným.
- Není třeba rozlišovat mezi příkazy a výrazy. Místo toho se používá jediný typ termů.
- Hodnoty proměnných se nemění. Po počátečním přiřazení hodnoty proměnné je možné v termu substituovat za výskyty dané proměnné příslušnou přiřazenou hodnotu.
- Termy se vyhodnocují postupným přepisováním a nikdy neobsahují volné proměnné.

Implementace separační logiky ze Software Foundations

Implementovaný jazyk je **dynamicky typovaný** (proměnné nemají staticky určen typ).

Do proměnných a buněk paměti je možné přiřazovat hodnoty různých typů:

- bool — booleovské hodnoty
- int — binárně reprezentovaná celá čísla
- loc — adresy paměti
- prim — primitivně definované funkce
- nerekurzivní a rekurzivní funkce — lambda výrazy (tj. de facto ukazatele na kód)
- další speciální typy hodnot — unit a hodnoty reprezentující neinicializované či chybné hodnoty

Implementace separační logiky ze Software Foundations

Datový typ val reprezentující hodnoty, které mohou být uloženy v buňkách paměti, přiřazeny do proměnněch, atd.:

```
Inductive val : Type :=
  | val_unit : val
  | val_bool : bool → val
  | val_int : int → val
  | val_loc : loc → val
  | val_prim : prim → val
  | val_fun : var → trm → val
  | val_fix : var → var → trm → val
  | val_uninit : val
  | val_error : val.
```

Implementace separační logiky ze Software Foundations

Datový typ prim reprezentuje funkce, které jsou přímo součástí jazyka (tj. nejsou definované pomocí lambda výrazů):

```
Inductive prim : Type :=
  | val_ref : prim          — alokace jedné buňky paměti
  | val_get : prim          — čtení z paměti (load)
  | val_set : prim          — zápis do paměti (store)
  | val_free : prim         — uvolnění jedné buňky paměti
  | val_add : prim          — sčítání
                ⋮
  | val_eq : prim           — test na rovnost (=)
  | val_le : prim           — menší nebo rovno (≤)
                ⋮
  | val_rand : prim         — generátor náhodných čísel
  | val_ptr_add : prim.     — ukazatelová aritmetika
```

Implementace separační logiky ze Software Foundations

Datový typ reprezentující **termy**:

```
Inductive trm : Type :=
  | trm_val : val → trm                          — hodnota
  | trm_var : var → trm                          — proměnná
  | trm_fun : var → trm → trm                    — nerekurzivní funkce
  | trm_fix : var → var → trm → trm              — rekurzivní funkce
  | trm_app : trm → trm → trm                    — aplikace funkce
  | trm_seq : trm → trm → trm                    — sekvence
  | trm_let : var → trm → trm → trm              — let ... in ...
  | trm_if : trm → trm → trm → trm               — if ... then ... else ...
```

Implementace separační logiky ze Software Foundations

Hoareovy trojice jsou zde tvaru:

$$\{P\}\ t\ \{\lambda r.\, Q\}$$

kde

- P, Q — assertions
- t — term
- r — výsledná hodnota termu