Formal Semantics of Programming Languages
Final examination

Instructor: 梁红瑾
Time: 08:00 ~ 09:50, 2024-12-25

Background (Part 1)

Programming Language. We examine a C/C++-style abstract programming language with the syntax shown in Figure 1. For simplicity, we do not distinguish numerals in syntax and real-life numbers.

$$
\begin{array}{rl}
(\textit{Numerals}) & n \\
(\textit{Variables}) & x \\
(\textit{Types}) & T ::= \textit{int} \mid \textit{int}* \\
(\textit{Expressions}) & e ::= n \mid x \mid *x \mid e_1 + e_2 \\
(\textit{Commands}) & c ::= T\ x \mid x = \textit{new int} \mid x = e \mid *x = e \mid \textit{free}(x) \mid \textit{skip} \mid c_1; c_2
\end{array}
$$

Figure 1: Syntax of the Programming Language

Configurations. The configurations of a program is of the shape (c, σ), where c is a command, and $\sigma = (s, h)$. Here s and h refers to the stack and heap memory respectively as shown in Figure 2. Locations are not numerals; we forbid the arithmetic operations on locations. *UNINIT* is a special value representing uninitialized memory.

$$
\begin{array}{rl}
(\textit{Locations}) & l \\
(\textit{Values}) & v ::= n \mid l \mid \textit{UNINIT} \\
(\textit{Stack}) & s \in \textit{Variables} \overset{\text{fin}}{\rightharpoonup} \textit{Values} \\
(\textit{Heap}) & h \in \textit{Locations} \overset{\text{fin}}{\rightharpoonup} (\textit{Numerals} \cup \{\textit{UNINIT}\})
\end{array}
$$

Figure 2: Definition of the configurations

Operational Semantics. The operational semantics of the programming language is defined in Figure 3 and Figure 4.

$$
\frac{}{(n, (s, h)) \Downarrow n}\ (\text{E-Num})
\qquad
\frac{s\ x = v}{(x, (s, h)) \Downarrow v}\ (\text{E-Var})
$$

$$
\frac{s\ x = l \qquad h\ l = n}{(*x, (s, h)) \Downarrow n}\ (\text{E-Deref})
\qquad
\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{(e_1 + e_2, (s, h)) \Downarrow (n_1 + n_2)}\ (\text{E-Add})
$$

Figure 3: Big-step operational semantics of expressions

$$
\frac{x \notin \mathrm{dom}(s)}{(T\ x, (s, h)) \to (\textit{skip}, (s \uplus \{x \rightsquigarrow \textit{UNINIT}\}, h))}\ (\text{E-NewVar})
\qquad
\frac{x \in \mathrm{dom}(s) \qquad l \notin \mathrm{dom}(h)}{(x = \textit{new int}, (s, h)) \to (\textit{skip}, (s\{x \rightsquigarrow l\}, h \uplus \{l \rightsquigarrow \textit{UNINIT}\}))}\ (\text{E-New})
$$

$$
\frac{x \in \mathrm{dom}(s) \qquad (e, (s, h)) \Downarrow v}{(x = e, (s, h)) \to (\textit{skip}, (s\{x \rightsquigarrow v\}, h))}\ (\text{E-Assign})
\qquad
\frac{s\ x = l \qquad l \in \mathrm{dom}(h) \qquad (e, (s, h)) \Downarrow v}{(*x = e, (s, h)) \to (\textit{skip}, (s, h\{l \rightsquigarrow v\}))}\ (\text{E-Mutate})
$$

$$
\frac{s\ x = l \qquad l \in \mathrm{dom}(h)}{(\textit{free}(x), (s, h)) \to (\textit{skip}, (s, h \setminus \{l\}))}\ (\text{E-Free})
\qquad
\frac{}{(\textit{skip}; c_2, (s, h)) \to (c_2, (s, h))}\ (\text{E-SkipSeq})
$$

$$
\frac{(c_1, (s, h)) \to ({c_1}', (s', h'))}{(c_1; c_2, (s, h)) \to ({c_1}'; c_2, (s', h'))}\ (\text{E-Seq})
$$

Figure 4: Small-step operational semantics of commands

Problem 1

Question.

The program c_0 is defined as

$$\mathit{int} *x; x = \mathit{new\ int}; *x = 42; \mathit{int} *y; y = x; \mathit{int}\ r; r = *y; \mathit{free}(y)$$

For (c_0, σ_0), write one of its full execution path, where $\sigma_0 = (\emptyset, \emptyset)$

Solution.

Problem 2

Question.

Let $\mathrm{terminate}(c, \sigma)$ denotes whether the command c terminates from the configuration (c, σ); that is, all of the execution paths from (c, σ) end at *skip*. Give the formal definition of $\mathrm{terminate}(c, \sigma)$.

Solution.

Problem 3

Question.

(a) Though there is no loop in the programming language, there are still commands that does not terminate. There exists command c_1 such that $\mathrm{terminate}(c_1, \sigma_0)$ does not hold, where $\sigma_0 = (\emptyset, \emptyset)$. Moreover, any execution path from (c_1, σ_0) does not end at *skip*. Construct such c_1 to satisfy the above requirements.

Solution.

Question.

(b) There exists command c_2 such that $\mathrm{terminate}(c_2, \sigma_0)$ does not hold, where $\sigma_0 = (\emptyset, \emptyset)$. Moreover, there exists one execution path from (c_2, σ_0) that ends at *skip*, and there exists another execution path that does not end at *skip*. Construct such c_2 to satisfy the above requirements.

Solution.

Background (Part 2)

Hence there is a need to define a type system to ensure the safety of the programming language, or to rule out commands that does not terminate. A type system is defined as follows. Firstly we define type contexts in Figure 5.

$$
\begin{array}{ll}
(\textit{Type Contexts}) & \Gamma \in \textit{Variables} \xrightharpoonup{\text{fin}} \textit{Partial Types} \\
(\textit{Partial Types}) & \tau ::= T \mid \lfloor T \rfloor \mid \lfloor \textit{int} \rfloor *
\end{array}
$$

Figure 5: Definition of type contexts

Then the type derivation rules are defined in Figure 6 and Figure 7.

$$
\frac{}{\Gamma \vdash n : \textit{int} \gg \Gamma}\ (\text{T-Num}) \qquad \frac{\Gamma(x) = \textit{int}}{\Gamma \vdash x : \textit{int} \gg \Gamma}\ (\text{T-Var-Copy})
$$

$$
\frac{\Gamma(x) = \textit{int}*}{\Gamma \vdash x : \textit{int}* \gg \Gamma\{x \rightsquigarrow \lfloor \textit{int}* \rfloor\}}\ (\text{T-Var-Move}) \qquad \frac{\Gamma(x) = \textit{int}*}{\Gamma \vdash *x : \textit{int} \gg \Gamma}\ (\text{T-Deref})
$$

$$
\frac{\Gamma \vdash e_1 : \textit{int} \gg \Gamma' \qquad \Gamma' \vdash e_2 : \textit{int} \gg \Gamma''}{\Gamma \vdash e_1 + e_2 : \textit{int} \gg \Gamma''}\ (\text{T-Add})
$$

Figure 6: Derivation rules for $\Gamma \vdash e : T \gg \Gamma'$

$$
\frac{x \notin \text{dom}(\Gamma)}{\Gamma \vdash T\ x \gg \Gamma\{x \rightsquigarrow \lfloor T \rfloor\}}\ (\text{T-NewVar}) \qquad \frac{\Gamma(x) = \lfloor \textit{int}* \rfloor}{\Gamma \vdash x = \textit{new int} \gg \Gamma\{x \rightsquigarrow \lfloor \textit{int} \rfloor *\}}\ (\text{T-New})
$$

$$
\frac{\Gamma \vdash e : T \gg \Gamma' \qquad (\Gamma'(x) = T) \vee (\Gamma'(x) = \lfloor T \rfloor)}{\Gamma \vdash x = e \gg \Gamma'\{x \rightsquigarrow T\}}\ (\text{T-Assign}) \qquad \frac{\Gamma \vdash e : \textit{int} \gg \Gamma' \qquad (\Gamma'(x) = \textit{int}*) \vee (\Gamma'(x) = \lfloor \textit{int} \rfloor *)}{\Gamma \vdash *x = e \gg \Gamma'\{x \rightsquigarrow \textit{int}*\}}\ (\text{T-Mutate})
$$

$$
\frac{(\Gamma'(x) = \textit{int}*) \vee (\Gamma'(x) = \lfloor \textit{int} \rfloor *)}{\Gamma \vdash \textit{free}(x) \gg \Gamma'\{x \rightsquigarrow \lfloor \textit{int}* \rfloor\}}\ (\text{T-Free}) \qquad \frac{}{\Gamma \vdash \textit{skip} \gg \Gamma}\ (\text{T-Skip})
$$

$$
\frac{\Gamma \vdash c_1 \gg \Gamma' \qquad \Gamma' \vdash c_2 \gg \Gamma''}{\Gamma \vdash c_1; c_2 \gg \Gamma''}\ (\text{T-Seq})
$$

Figure 7: Derivation rules for $\Gamma \vdash c \gg \Gamma'$

Problem 4

Question.

The program c_0 is defined as

$$\mathit{int} *x; x = \mathit{new\ int}; *x = 42; \mathit{int} *y; y = x; \mathit{int}\ r; r = *y; \mathit{free}(y)$$

Write down a type context Γ such that $\emptyset \vdash c_0 \gg \Gamma$, and give its type derivation.

Solution.

Problem 5

Question.

The type system is incomplete; that is, there exists command c such that $\mathrm{terminate}(c, \sigma_0)$ holds, where $\sigma_0 = (\emptyset, \emptyset)$; however, $\neg\exists\Gamma.(\emptyset \vdash c \gg \Gamma)$. Construct such c to satisfy the requirement.

Solution.

Problem 6

Question.

(a) Consider Property 1 given by

$$\forall c.\ ((\exists\Gamma.\ \emptyset \vdash c \gg \Gamma \wedge \mathrm{freed}(\Gamma)) \Rightarrow \mathrm{noMemoryLeak}(c)) \tag{1}$$

where

$$\mathrm{freed}(\Gamma) \stackrel{\mathrm{def}}{=} \forall x \in \mathrm{dom}(\Gamma).\ \Gamma(x) = \lfloor T \rfloor \vee \Gamma(x) = \mathit{int}$$

$$\mathrm{noMemoryLeak}(c) \stackrel{\mathrm{def}}{=} \forall s, h.\ ((c, (\emptyset, \emptyset)) \rightarrow^* (\mathit{skip}, (s, h)) \Rightarrow h = \emptyset)$$

Give a counterexample to the Property 1; that is, construct a command c such that $\exists\Gamma.\ \emptyset \vdash c \gg \Gamma \wedge \mathrm{freed}(\Gamma)$ holds, but $\mathrm{noMemoryLeak}(c)$ does not hold.

Solution.

Question.

(b) Adjust the derivation rules in the typing system to make the Property 1 hold. You don't need to prove this.

Solution.