1 Ownership and moving

We formalize a 'move only' semantics. Here, we can only assign variables. After assignment, ownership of resource is moved to that variable. Borrowing and mutable references are disallowed.

1.1 Syntax

$$S ::= \texttt{skip} \mid S_1; S_2 \mid \texttt{let}\ x : \tau\ \texttt{in}\ S' \mid x = e$$

$$e ::= x \mid i \mid e_1 + e_2$$

$$\tau ::= \texttt{Int}$$

Rust syntax `let x = e` is desugared as `let x in (x = e)` to distinguish variable declaration `x` from assignment of a value to the resource `x` owns. Types added for clarity in borrowing.

1.2 Semantics

The semantic domain is $\mathbb{Z}_{ext} := \mathbb{Z} \cup \{\bot, -\}$, so variables can hold integers values, be non-declared ($-$) or not assigned ($\bot$). In particular, **Var** is the set of all variables, **Num** is the set of all numbers, **Add** denotes the set of all pairs from **Exp** and $\textbf{Exp} = \textbf{Num} \cup \textbf{Var} \cup \textbf{Add}$. We have semantic functions $\mathcal{V} : \textbf{Exp} \to \mathcal{P}(\textbf{Var})$ gathering all variables in an expression and $\mathcal{N}$: $\textbf{Num} \to \mathbb{Z}$ translating syntactic numbers to mathematical numbers.

Memory is represented by a state function $s : \textbf{Var} \to \mathbb{Z}_{ext}$. **State** is the set of all possible memory arrangements. An evaluation function $\mathcal{A} : \textbf{Exp} \times \textbf{State} \to \mathbb{Z}_{ext}$ takes into account memory in the interpretation of arithmetic expressions. Note that to perform addition we need each addend to be evaluated to an integer. Otherwise the result is undefined ($-$).

1.2.1 Big step semantics

[skip$_{\text{ns}}$] $\quad \langle \texttt{skip}, s\rangle \to s$

[comp$_{\text{ns}}$] $\quad \dfrac{\langle S_1, s\rangle \to s' \qquad \langle S_2, s'\rangle \to s''}{\langle S_1; S_2, s\rangle \to s''}$

[let$_{\text{ns}}$] $\quad \dfrac{\langle S, s[x \mapsto \bot]\rangle \to s'}{\langle \texttt{let}\ x : \tau\ \texttt{in}\ S, s\rangle \to s'[x \mapsto s(x)]}$

[ass$_{\text{ns}}$] $\quad \langle x = e, s\rangle \to s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -] \qquad$ if $\mathcal{A}[\![x]\!]s = \bot$, $\mathcal{A}[\![e]\!]s \neq \bot$ and $\mathcal{A}[\![e]\!]s \neq -$

where $\mathcal{V}(e) \mapsto -$ means $\forall x \in \mathcal{V}(e)$, $x \mapsto -$.

The rule for `let` states that a `let`-statement should only declare the variable x as $\bot$. When the body of the let-statement is finished, x should recover its value before the `let`-statement. For assignment, there are several side-conditions. The first states that `x` must be only declared. The second and third, state that the expression must result in a value.

We have formalized the following properties:

Theorem 1.1 (Determinism)
$\langle S, s\rangle \to s' \wedge \langle S, s\rangle \to s'' \implies s' = s''$.

Theorem 1.2 (Variable allocation)
$\langle S, s\rangle \to s' \implies (\forall y. \mathcal{A}[\![y]\!]s = - \implies \mathcal{A}[\![y]\!]s' = -)$.

After termination, a program leaves no variables in memory.

1.2.2 Small step semantics

The small steps semantics operates on program instructions:

$$I ::= S \mid (x, v)$$

where $x \in$**Var**, $v \in \mathbb{Z}_{ext}$. The added command stores in the stack the reset operations produced by let.

$$
\begin{array}{ll}
[\text{load}_{\text{sos}}] & \langle \texttt{skip}, I :: L, s\rangle \Rightarrow \langle I, L, s\rangle \\
[\text{comp}_{\text{sos}}] & \langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle \\
[\text{ass}_{\text{sos}}] & \langle x = e, L, s\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]\rangle \\
[\text{let}_{\text{sos}}] & \langle \texttt{let}\ x : \tau\ \texttt{in}\ S, L, s\rangle \Rightarrow \langle S, (x, s(x)) :: L, s[x \mapsto \perp]\rangle \\
[\text{set}_{\text{sos}}] & \langle (x, v), L, s\rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto v]\rangle
\end{array}
$$

We formalized the following properties:

Lemma 1.3 (Break composition)
$\langle S_1; S_2, L, s\rangle \Rightarrow^* \langle \textit{skip}, L, s'\rangle \implies$
$\exists s''. \langle S_1; S_2, L, s\rangle \Rightarrow \langle S_1, S_2 :: L, s\rangle \Rightarrow^*$
$\Rightarrow^* \langle \textit{skip}, S_2 :: L, s''\rangle \Rightarrow \langle S_2, L, s''\rangle \Rightarrow^* \langle \textit{skip}, L, s'\rangle$

Lemma 1.4 (Break let)
$\langle \textit{let}\ x : \tau\ \textit{in}\ S, L, s\rangle \Rightarrow^* \langle \textit{skip}, L, s'\rangle \implies$
$\exists s''. \langle \textit{let}\ x : \tau\ \textit{in}\ S, L, s\rangle \Rightarrow^* \langle \textit{skip}, (x, s(x)) :: L, s''\rangle \Rightarrow$
$\Rightarrow \langle (x, s(x)), L, s''\rangle \Rightarrow \langle \textit{skip}, L, s'\rangle$

Proposition 1.5 (Stack discipline)
$\langle S, L', s\rangle \Rightarrow^* \langle \textit{skip}, L', s'\rangle \implies (\forall L. \langle S, L, s\rangle \Rightarrow^* \langle \textit{skip}, L, s'\rangle)$

Proposition 1.6 (Sequentiality)
$\langle S_1, L, s\rangle \Rightarrow^* \langle \textit{skip}, L, s\rangle \implies \langle S_1; S_2, L, s\rangle \Rightarrow^* \langle S_2, L, s\rangle$

Theorem 1.7 (Determinism)
$\langle S, L, s\rangle \Rightarrow^* \langle \textit{skip}, L, s'\rangle \wedge \langle S, L, s\rangle \Rightarrow^* \langle \textit{skip}, L, s''\rangle \implies s' = s''$.

1.2.3 Compile time check

Big step semantics is equivalent to small step semantics (which dropped side conditions for assignment) plus a compile time check of these dropped conditions. The check performs no computation and uses an abstract *reduced state* r : **Var** $\rightarrow \{-, \perp, \star\}$ where $\star$ represents an unspecified concrete value and **RState** is the set of reduced states. To each state, corresponds an abstract reduced state replacing concrete integers by $\star$. We say both states are *related*. Here are the rules of the *compile time checker*:

$$
\begin{aligned}
[\texttt{skip}, \texttt{Nil}, r] &\rightarrow \texttt{true} \\
[\texttt{skip}, P :: L, r] &\rightarrow [P, L, r] \\
[S_1; S_2, L, r] &\rightarrow [S_1, S_2 :: L, r] \\
[x = e, L, r] &\rightarrow [\texttt{skip}, L, r[x \mapsto \star][\mathcal{V}(e) \mapsto -]] \\
&\quad \text{if } r(x) = \perp \text{ and } \forall y \in \mathcal{V}(e), r(y) = \star \\
&\rightarrow \texttt{false} \text{ otherwise} \\
[\texttt{let } x : \tau \texttt{ in } S, L, r] &\rightarrow [S, (x, r(x)) :: L, r[x \mapsto \perp]] \\
[(x, v), L, r] &\rightarrow [\texttt{skip}, L, r[x \mapsto v]]
\end{aligned}
$$

We formalized the following properties:

Theorem 1.8 (Termination)
The compile checker always terminates.

Theorem 1.9 (Semantic equivalence)
$\langle S, s\rangle \rightarrow s' \iff \exists L. \langle S, L, s\rangle \Rightarrow^* \langle \textit{skip}, L, s'\rangle \wedge [S, L, \textit{reduced } s] \rightarrow^* \textit{true}$

1.3 Safety

Safety proofs can be given following Wright and Felleisen (1994).

Theorem 1.10 (Preservation)
$[S, L, \textit{reduced } s] \rightarrow^* \textit{true} \wedge \langle S, L, s\rangle \Rightarrow \langle S', L', s'\rangle \implies$
$[S', L', \textit{reduced } s'] \rightarrow^* \textit{true}.$

Theorem 1.11 (Progress)
$[S, L, r] \rightarrow^* \textit{true} \implies$
$S = \textit{skip} \wedge L = \textit{Nil} \vee \forall \textit{concrete } r. \exists S', L', s'. \langle S, L, s\rangle \Rightarrow \langle S', L', s'\rangle$

References

Wright, A. K. and Felleisen, M. (1994). A syntactic approach to type soundness. *Information and computation*, 115(1):38–94.