1 Ownership and moving

We formalize a 'move only' semantics Wessel et al. (2019).

- 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 $\mathbb{Z}_{ext} := \mathbb{Z} \cup \{\perp, -\}$, so variables can hold integers values, be non-declared ($-$) or not assigned ($\perp$). 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 **Exp** = **Num** $\cup$ **Var** $\cup$ **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

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

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

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

$[\text{ass}_{ns}]$ $\quad \langle x = e, s \rangle \to s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -]$ $\qquad$ if $\mathcal{A}[\![x]\!]s = \perp$, $\mathcal{A}[\![e]\!]s$ $\neq \perp$ 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 $x \in \mathbf{Var}$, $v \in \mathbb{Z}_{ext}$. The added command stores in the stack the reset operations produced by let.

$$[\text{load}_{\text{sos}}] \qquad \langle \texttt{skip}, I :: L, s \rangle \Rightarrow \langle I, L, s \rangle$$

$$[\text{comp}_{\text{sos}}] \qquad \langle S_1; S_2, L, s \rangle \Rightarrow \langle S_1, S_2 :: L, s \rangle$$

$$[\text{ass}_{\text{sos}}] \qquad \langle x = e, L, s \rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto \mathcal{A}[\![e]\!]s][\mathcal{V}(e) \mapsto -] \rangle$$

$$[\text{let}_{\text{sos}}] \qquad \langle \texttt{let } x : \tau \texttt{ in } S, L, s \rangle \Rightarrow \langle S, (x, s(x)) :: L, s[x \mapsto \bot] \rangle$$

$$[\text{set}_{\text{sos}}] \qquad \langle (x, v), L, s \rangle \Rightarrow \langle \texttt{skip}, L, s[x \mapsto v] \rangle$$

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 : $\mathbf{Var} \to \{-, \bot, \star\}$ where $\star$ represents an unspecified concrete value and $\mathbf{RState}$ 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*:

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

We formalized the following properties:

Theorem 1.8 (Termination)
The compile checker always terminates.

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

1.3   Safety

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

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

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

References

Wessel, N., Wiedijk, F., and Schoolderman, M. (2019). The semantics of ownership and borrowing in the rust programming language.

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