1 The Programming Language

Below shows the syntax of the programming language we consider, which is an extension of the one in the original Iris paper. We also take Rust language as a reference in the design of the language.

$$
\begin{aligned}
v \in \mathit{Val} &::= () \mid z \mid \texttt{true} \mid \texttt{false} \mid l \mid \lambda x.e \mid \cdots \\
e \in \mathit{Expr} &::= v \mid x \mid e_1(e_2) \mid \texttt{ref}(e) \mid !e \mid e_1 \leftarrow e_2 \mid \texttt{fork}\{e\} \mid e_1 \,;; \, e_2 \mid \texttt{loop}_e\, e \\
&\quad \mid \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \mid \texttt{break}\, e \mid \texttt{continue} \mid \texttt{call}\, e \mid \texttt{return}\, e \mid \cdots \\
K \in \mathit{Ctx} &::= \bullet \mid K(e) \mid v(K) \mid \texttt{ref}(K) \mid !K \mid K \leftarrow e \mid v \leftarrow K \mid K \,;; \, e \mid \texttt{loop}_e\, K \\
&\quad \mid \texttt{if}\, K\, \texttt{then}\, e_2\, \texttt{else}\, e_3 \mid \texttt{break}\, K \mid \texttt{call}\, K \mid \texttt{return}\, K \mid \cdots
\end{aligned}
$$

To allow context binding, we use $\texttt{loop}_e\, e$ to represent a loop statement with e as its loop body. e serves as a parameter in the corresponding evaluation context for the loop command. After the completion of evaluation for expression in K, we can still recover the complete loop body from the subscription of $\texttt{loop}_e$. We can wrap the conventional loop definition as $\texttt{loop}\, e \triangleq \texttt{loop}_e\, e$.

The `break` statement and `continue` statement are scoped by $\texttt{loop}_e$ and can only manipulate the control flow within the loop. The sequence statement and if statement definitions are standard.

We also consider control flow related to function invocations. To the resolve the problem of whether the `return` statement can penetrate unnamed functions defined with the λ operator (e.g. which one of 1 and 2 should $(\lambda x.\texttt{return}\, x)(1) \,;; \, 2$ evaluate to), we use `call` to scope the influence of `return`. We can still define recursive functions in lambda calculus with the `return` functioning properly.

Following is an example defining the factorial function, where we early return 1 when n is 0 and follows it by an invocation to f recursively. The expression $\texttt{call}\,(\mathit{FACT}(n))$ would return the value of $n!$. Removing `call` in this definition will cause the evaluation of $\mathit{FACT}(n)$ always goes into $\texttt{return}\, 1$.

$$
\begin{aligned}
\mathit{FACT}' &\triangleq \lambda f.\lambda n.(\texttt{if}\, n = 0\, \texttt{then}\, \texttt{return}\, 1) \,;; \, n \times (\texttt{call}\,(f\, f\, (n-1))) \\
\mathit{FACT} &\triangleq \mathit{FACT}'\, \mathit{FACT}'
\end{aligned}
$$

2 Operational Semantics

We then define the operational semantics of the language.

$$
\begin{aligned}
(\texttt{loop}_e\, v, \sigma) &\to_h (\texttt{loop}_e\, e, \sigma, \epsilon) \\
(\texttt{loop}_e\, (\texttt{continue}), \sigma) &\to_h (\texttt{loop}_e\, e, \sigma, \epsilon) \\
(\texttt{loop}_e\, (\texttt{break}\, v), \sigma) &\to_h (v, \sigma, \epsilon) \\
(K[\texttt{break}\, v], \sigma) &\to_h (\texttt{break}\, v, \sigma, \epsilon) \qquad \text{if } K \neq \bullet \text{ and } K \in \mathrm{PenCtx}(\texttt{break}) \\
(K[\texttt{continue}], \sigma) &\to_h (\texttt{continue}, \sigma, \epsilon) \qquad \text{if } K \neq \bullet \text{ and } K \in \mathrm{PenCtx}(\texttt{continue})
\end{aligned}
$$

Figure 1: Head Reductions for Loop

Figure 1 shows head reductions for $\texttt{loop}_e$, `break`, and `continue`.

- The first two rules says that when the evaluation of the loop body finishes or it encounters a `continue`, it reset the expression in the hole to e, the complete loop body. We no longer require the body evaluate to unit, and such requirement can be met by definition $\texttt{loop}\, e \triangleq \texttt{loop}_e\, (e; ; ())$.
- The third rule says when the loop encounters a `break` in the hole, the evaluation for the entire loop finishes with the value carried by `break` as its result.
- The last two rules says `break` and `continue` can terminate the reduction of their evaluation context K, if K is their penetrable context PenCtx defined as:

$$
\begin{aligned}
\mathrm{PenCtx}(\texttt{break}) &\triangleq \mathit{Cxt} \setminus \texttt{loop}_e \setminus \texttt{call} \\
\mathrm{PenCtx}(\texttt{continue}) &\triangleq \mathit{Cxt} \setminus \texttt{loop}_e \setminus \texttt{call}
\end{aligned}
$$

where $Cxt \backslash \texttt{loop}_e \backslash \texttt{call}$ represents the set of contexts without using the $\texttt{loop}_e$ and `call` constructs. This limits effects of `break` and `continue` to stay within the loop body and function body. We forbid K to be $\bullet$ to avoid infinite reductions.

$$\begin{aligned}
(\texttt{call}\, v, \sigma) &\to_h (v, \sigma, \epsilon) \\
(\texttt{call}\, (\texttt{return}\, v), \sigma) &\to_h (v, \sigma, \epsilon) \\
(K[\texttt{return}\, v], \sigma) &\to_h (\texttt{return}\, v, \sigma, \epsilon) \quad \text{if } K \neq \bullet \text{ and } K \in \text{PenCtx}(\texttt{return})
\end{aligned}$$

Figure 2: Head Reductions for Function Invocation

Figure 2 shows head reductions for `call` and `return`.

- The first two rules says that when the function body evaluates to a value or a `return` statement, the entire function invocation statement evaluates to the corresponding value.
- Similar to `break` and `continue`, `return` can terminate the evaluation of its penetrable context:

  $$\text{PenCtx}(\texttt{return}) \triangleq Cxt \backslash \texttt{call}$$

  which limits `return` to affect control flow within the innermost `call`.
- The $\texttt{return}\, v$ itself can not reduce to v outside a function invocation to allow the adequacy theorem hold for our weakest precondition definition, which seems reasonable if we consider any program consists of only an invocation to the main function.

Remark. Notice that there are 4 types of terminals for an expression in the reduction: v, $\texttt{break}\, v$, `continue`, and $\texttt{return}\, v$. They cannot be further reduced and correspond to 4 different postconditions in the weakest precondition we defined later.

$$\text{terminals} \triangleq Val \cup \{\texttt{break}\, v, \texttt{continue}, \texttt{return}\, v \mid v \in Val\}$$

$$\begin{aligned}
(\texttt{if true then}\, e_2\, \texttt{else}\, e_3, \sigma) &\to_h (e_2, \sigma, \epsilon) \\
(\texttt{if false then}\, e_2\, \texttt{else}\, e_3, \sigma) &\to_h (e_3, \sigma, \epsilon) \\
(v\, ;;\, e, \sigma) &\to_h (e, \sigma, \epsilon) \\
(\texttt{fork}\{e\}, \sigma) &\to_h ((), \sigma, e) \quad \text{if } e \text{ is well-formed} \\
&\cdots
\end{aligned}$$

Figure 3: Head Reductions for Other Expressions

Figure 3 shows the head reduction of the if statement, the sequence statement, and the fork statement. First three rules are standard, while the last one for fork is modified from the one in the original Iris paper, which requires the expression in the forked thread to be well-formed.

Definition 1. *An expression is well-formed if and only if every* $\texttt{break}$ *and* $\texttt{continue}$ *in it is scoped by a* $\texttt{loop}_e$ *and every* $\texttt{return}$ *in it is scoped by a* $\texttt{call}$.

3 Weakest Precondition and Proof Rules

We define the weakest precondition with multiple postconditions for different exit type as follows:

$$\begin{aligned}\mathsf{wp}\, e\, \{\Phi_N\}\, \{\Phi_B\}\, \{\Phi_C\}\, \{\Phi_R\} \triangleq\ & (e \in \mathit{Val} \land \dot{\Rightarrow} \Phi_N(e)) \\ & \lor (\exists v \in \mathit{Val}. e = \mathtt{break}\, v \land \dot{\Rightarrow} \Phi_B(v)) \\ & \lor (e = \mathtt{continue} \land \dot{\Rightarrow} \Phi_C()) \\ & \lor (\exists v \in \mathit{Val}. e = \mathtt{return}\, v \land \dot{\Rightarrow} \Phi_R(v)) \\ & \lor \Big(\forall \sigma. e \notin \text{terminals} \land S(\sigma) \mathrel{-\!\!*} \dot{\Rightarrow} \big(\text{red}(e, \sigma) \\ & \quad \land \triangleright \forall e', \sigma', \vec{e}_f. \big((e, \sigma) \to_t (e', \sigma', \vec{e}_f)\big) \mathrel{-\!\!*} \dot{\Rightarrow} \\ & \quad \big(S(\sigma') * \mathsf{wp}\, e' \{\Phi_N\} \{\Phi_B\} \{\Phi_C\} \{\Phi_R\} * \divideontimes_{e' \in \vec{e}_f} \mathsf{wp}\, e' \{\top\} \{\top\} \{\top\} \{\top\}\big)\big) \Big)\end{aligned}$$

There are 5 cases in total:

- **Normal exit:** If the expression has evaluated to a value, then Φ_N should hold.
- **Break exit:** If the expression has evaluated to `break`, then Φ_B should hold.
- **Continue exit:** If the expression has evaluated to `continue`, then Φ_C should hold.
- **Return exit:** If the expression has evaluated to `return`, then Φ_R should hold.
- **Preservation:** If the expression is a non-terminal, then for any program state, the expression is reducible, and after one step of reduction, the weakest precondition holds for the new expression and any forked thread can terminate normally (where the well-formedness come into effect).

$$\Phi_B(v) \vdash \mathsf{wp}\, \mathtt{break}\, v\, \{\bot\}\, \{\Phi_B\}\, \{\bot\}\, \{\bot\} \qquad \text{(WP-BREAK)}$$

$$\Phi_C() \vdash \mathsf{wp}\, \mathtt{continue}\, \{\bot\}\, \{\bot\}\, \{\Phi_C\}\, \{\bot\} \qquad \text{(WP-CONTINUE)}$$

$$\Phi_R(v) \vdash \mathsf{wp}\, \mathtt{return}\, v\, \{\bot\}\, \{\bot\}\, \{\bot\}\, \{\Phi_R\} \qquad \text{(WP-RETURN)}$$

$$\square(I \mathrel{-\!\!*} \mathsf{wp}\, e\, \{\_.I\}\, \{\Phi_B\}\, \{\_.I\}\, \{\Phi_R\}) * I \vdash \mathsf{wp}\, (\mathtt{loop}_e\, e)\, \{\Phi_B\}\, \{\bot\}\, \{\bot\}\, \{\Phi_R\} \qquad \text{(WP-LOOP)}$$

$$\begin{aligned}&(v = \mathtt{true} \mathrel{-\!\!*} \mathsf{wp}\, e_2\, \{\Phi_N\}\, \{\Phi_B\}\, \{\Phi_C\}\, \{\Phi_R\}) \\ \land &(v = \mathtt{false} \mathrel{-\!\!*} \mathsf{wp}\, e_3\, \{\Phi_N\}\, \{\Phi_B\}\, \{\Phi_C\}\, \{\Phi_R\})\end{aligned} \vdash \mathsf{wp}\, \mathtt{if}\, v\, \mathtt{then}\, e_2\, \mathtt{else}\, e_3\, \{\Phi_N\}\, \{\Phi_B\}\, \{\Phi_C\}\, \{\Phi_R\} \qquad \text{(WP-IF)}$$

$$\begin{aligned}&\mathsf{wp}\, e_1\, \{\_.\mathsf{wp}\, e_2\, \{\Phi_N\}\, \{\Phi_B\}\, \{\Phi_C\}\, \{\Phi_R\}\} \\ &\quad \{\Phi_B\}\, \{\Phi_C\}\, \{\Phi_R\}\end{aligned} \vdash \mathsf{wp}\, e_1\, ;;\, e_2\, \{\Phi_N\}\, \{\Phi_B\}\, \{\Phi_C\}\, \{\Phi_R\} \qquad \text{(WP-SEQ)}$$

$$\mathsf{wp}\, e\, \{\Phi\}\, \{\bot\}\, \{\bot\}\, \{\Phi\} \vdash \mathsf{wp}\, \mathtt{call}\, e\, \{\Phi\}\, \{\bot\}\, \{\bot\}\, \{\bot\} \qquad \text{(WP-CALL)}$$

$$\begin{aligned}&\mathsf{wp}\, e\, \{v.\mathsf{wp}\, K[v]\, \{\Phi_N\}\, \{\Phi_B\}\, \{\Phi_C\}\, \{\Phi_R\}\} \\ &\quad \{v.K \in \text{PenCtx}(\mathtt{break}) \land \Phi_B(v)\} \\ &\quad \{v.K \in \text{PenCtx}(\mathtt{continue}) \land \Phi_C(v)\} \\ &\quad \{v.K \in \text{PenCtx}(\mathtt{return}) \land \Phi_R(v)\}\end{aligned} \vdash \mathsf{wp}\, K[e]\, \{\Phi_N\}\, \{\Phi_B\}\, \{\Phi_C\}\, \{\Phi_R\} \qquad \text{(WP-BIND)}$$

$$\ldots$$

Figure 4: Some Rules for Weakest Precondition

Figure 4 shows some rules derivable from our definition of the weakest precondition, which we have proved informally and should hold if formalized in Coq using Iris. We mainly demonstrate rules related to the control flow and binding.

- The first three rules, WP-BREAK, WP-CONTINUE, and WP-RETURN, are rules for three primitives to manipulate control flow.
- The WP-LOOP is the standard loop rule involving loop invariant. The WP-IF and WP-SEQ are also standard rules for if statement and sequence statement with multiple exit conditions.
- The WP-CALL allows conversion from return postconditions to normal postconditions upon the completion of function invocation.
- The most interesting rule is the WP-BIND rule, which is the main obstacle in the previous effort to equip Iris with control flow. In the weakest precondition of inner expression e, the normal postconditions is simply the weakest precondition for the remaining context K. The postconditions for other exit kind is more interesting. Taking `break` for example, the postcondition says the remaining context K is penetrable by `break` and the overall break condition Φ_B holds.
  - If K is penetrable, then the `break` inside e can also break the control flow in K. Thus the break postconditions of e and $K[e]$ should be the same.
  - If K is impenetrable, then the `break` inside e is scoped by some $\texttt{loop}_e$ in K. Therefore, we should forbid the break condition of e to affect the break condition of $K[e]$ (which is determined by statements outside the $\texttt{loop}_e$ in K) by evaluating $K \in \text{PenCtx}(\texttt{break})$ and the entire break condition to $\bot$. Consequently, the weakest precondition in the premise is invalidated (because e actually has a break condition).
  - In the second case, to handle the break condition of e properly, we should use WP-BIND to bind the context outside the loop and apply WP-LOOP to the entire loop containing e.

4 Typing

We also think of the possible typing system of the language consisting of following primary types.

$$\tau, A, B \in \mathit{Type} ::= \texttt{unit} \mid \texttt{int} \mid \texttt{bool} \mid A \to B \mid \cdots$$

With multiple evaluation terminals, we should type the result of all possible kinds of terminal for an expression.

4.1 Syntactic Typing

We first define the notation for syntactic type as:

$$\Gamma \vdash e : A_N, A_B, A_C, A_R$$

It states that given type assignment Γ, if the expression evaluates to a value, a break terminal, a continue terminal, or a return terminal, then types of the value or values carried by terminals are respectively A_N, A_B, A_C, and A_R. If the expression evaluation will not reach some terminals, we can type corresponding values arbitrarily using a variable τ. For example, $\Gamma \vdash \texttt{break}\,1 : \tau, \texttt{int}, \tau, \tau$ means that the break type of the expression is `int`, while other terminal types are arbitrary and we can set these types manually in favor of our proof.

Figure 5 shows some typing rules related to control flows and expression application.

- First three rule types three new terminal expressions. Notice that we could have nested break/continue/return structures in `break` and `return` expressions. The continue type of any expression will always be `unit` in this language, but we still use a variable A_C to represent it in most cases.
- In TYPE-LOOP rule, The break type of the loop body is converted into the normal type of the loop. We require the normal type and continue type of the loop body to be `unit`, which is necessary in the Rust language.
- In TYPE-CALL rule, the normal type and return type of function body should agree. Also, the function body should have τ as its break type and continue type to guarantee well-formedness.
- TYPE-SEQ and TYPE-IF rules are straight forward, where break type, continue type, and return type of expressions should agree respectively.
- The TYPE-APPLY rule, in contrast to TYPE-CALL, allow the control flow types (A_C, A_B, A_R) in the lambda term e_1 to be exposed after the application.

$$\text{TYPE-BREAK}\ \frac{\Gamma \vdash e : A_B, A_B, A_C, A_R}{\Gamma \vdash \texttt{break}\, e : \tau, A_B, A_C, A_R} \qquad \text{TYPE-CONTINUE}\ \Gamma \vdash \texttt{continue} : \tau, \tau, \texttt{unit}, \tau$$

$$\text{TYPE-RETURN}\ \frac{\Gamma \vdash e : A_R, A_B, A_C, A_R}{\Gamma \vdash \texttt{return}\, e : \tau, A_B, A_C, A_R}$$

$$\text{TYPE-LOOP}\ \frac{\Gamma \vdash e : \texttt{unit}, A_B, \texttt{unit}, A_R}{\Gamma \vdash \texttt{loop}_e\, e : A_B, \tau, \tau, A_R} \qquad \text{TYPE-CALL}\ \frac{\Gamma \vdash e : A_R, \tau, \tau, A_R}{\Gamma \vdash \texttt{call}\, e : A_R, \tau, \tau, \tau}$$

$$\text{TYPE-SEQ}\ \frac{\Gamma \vdash e_1 : A'_N, A_B, A_C, A_R \qquad \Gamma \vdash e_2 : A_N, A_B, A_C, A_R}{\Gamma \vdash e_1 \,;; \, e_2 : A_N, A_B, A_C, A_R}$$

$$\text{TYPE-IF}\ \frac{\Gamma \vdash e_1 : \texttt{bool}, A_B, A_C, A_R \qquad \Gamma \vdash e_2 : A_N, A_B, A_C, A_R \qquad \Gamma \vdash e_3 : A_N, A_B, A_C, A_R}{\Gamma \vdash \texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3 : A_N, A_B, A_C, A_R}$$

$$\text{TYPE-APPLY}\ \frac{\Gamma \vdash e_1 : A'_N \to (A_N, A_B, A_C, A_R) \qquad \Gamma \vdash e_2 : A'_N, A_B, A_C, A_R}{\Gamma \vdash e_1(e_2) : A_N, A_B, A_C, A_R}$$

$$\text{where } A \to (B_1, B_2, \cdots) \triangleq A \to B_1, A \to B_2, \cdots$$

Figure 5: Some Typing Rules

4.2 Semantic Typing

We can also do semantic typing for the language, and with the definition of weakest precondition, we can easily derive the following definition.

$$\Gamma \vDash e : A_N, A_B, A_C, A_R \triangleq \Gamma \vDash \{\text{True}\}\, e\, \{\mathbf{V}[\![A_N]\!]\}\, \{\mathbf{V}[\![A_B]\!]\}\, \{\mathbf{V}[\![A_C]\!]\}\, \{\mathbf{V}[\![A_R]\!]\}$$

This definition is quite informal with some notation abuse, but its meaning and some skipped notation definitions are similar to those in the Tutorial on Proving Semantic Type Soundness in Iris.