UCSD CSE131 F19 – Copperhead

October 23, 2019

Due Date: 11pm Wednesday, October 23 **Open to Collaboration**
You will implement Copperhead, a language like Boa extended with a static type system, variables, and while loops.
Classroom: `https://classroom.github.com/a/y0UCOlDY` Github: `http://github.com/ucsd-cse131-f19/pa3-student`

Syntax

The concrete syntax and type language for Copperhead is below. We use $\cdots$ to indicate *one or more* of the previous element. So a while expression has at least two sub-expressions, and a let expression at least one binding and body expression.

e	:=	n | `true` | `false`			
	|	`(let ((`x e`) ` $\cdots$`) `$e\cdots$`)`			
	|	`(if` e e e`)`	τ	:=	Num | Bool
	|	`(`op_2 e e`)` | `(`op_1 e`)`	Γ	:=	$\{x : \tau, \cdots\}$
	|	`(while` e $e\cdots$`)` | `(set` x e`)`	$\Gamma[x]$	*means*	*look up the type of* x *in* Γ
op_1	:=	`add1` | `sub1` | `isNum` | `isBool`	$(x, \tau) :: \Gamma$	*means*	*add* x *to* Γ *with type* τ
op_2	:=	`+` | `-` | `*` | `<` | `>` | `==`	$\Gamma \vdash e : \tau$	*means*	*in environment* Γ, e *has type* τ
n	:=	63-bit signed number literals			
x	:=	variable names			

Semantics

The behavior of the existing forms is largely same as in Boa, with a few modifications:

- `let` expressions in Copperhead can have multiple body expressions. These expressions should be evaluated in order, and the result of the let expression is the result of the last expression in the body.
- The **static type checker** for Copperhead eliminates the need to check tags of operands to binary operators like `+`. Overflow is checked and reported at runtime as in Boa.
- The built-in variable `input` must always be a number. If a user gives a non-number value, the runtime should report a dynamic error that includes the string `"input must be a number"`.

The existing static errors (duplicate bindings and unbound identifiers) should be reported as in Boa.
There are two new expressions in Copperhead, as well:

- `set` expressions update the value of a variable. The behavior of a `set` expression is to evaluate its subexpression, then set the value of the named variable to the result of that subexpression. The result of the entire `set` expression is the new value, and its effect is to make future accesses of that variable get the updated value.
- `while` expressions evaluate a condition and body repeatedly. The condition expression is the first one that appears in the while expression – it should evaluate to a boolean (the type rules below enforce this), and if it evaluates to `true` the body expressions evaluate in order. This process is repeated until the condition evaluates to `false`, the body is not executed, and the entire `while` expression evaluates to `false`.

Type Checking

Copperhead has the typing rules shown in figure 1. If an expression cannot be typed according to these rules, the compiler should report a static error containing `"Type mismatch"`. A few rules benefit from some extra explanation:

- TR-Sequence describes type-checking a sequence of expressions, as found in the body of while and let expressions. To type-check a sequence, all the expressions must type to *some* type, though they can be different across expressions. The type of the entire sequence is the type of the *last* expression, τ_n.

$$\text{TR-Num}\quad n : \mathsf{Num} \qquad \text{TR-True}\quad \texttt{true} : \mathsf{Bool} \qquad \text{TR-False}\quad \texttt{true} : \mathsf{Bool}$$

$$\text{TR-Plus}\ \frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash \texttt{(+ } e_1\ e_2\texttt{)} : \mathsf{Num}} \qquad \text{TR-Minus}\ \frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash \texttt{(- } e_1\ e_2\texttt{)} : \mathsf{Num}}$$

$$\text{TR-Times}\ \frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash \texttt{(* } e_1\ e_2\texttt{)} : \mathsf{Num}}$$

$$\text{TR-Add1}\ \frac{\Gamma \vdash e : \mathsf{Num}}{\Gamma \vdash \texttt{(add1 } e\texttt{)} : \mathsf{Num}} \qquad \text{TR-Sub1}\ \frac{\Gamma \vdash e : \mathsf{Num}}{\Gamma \vdash \texttt{(sub1 } e\texttt{)} : \mathsf{Num}}$$

$$\text{TR-IsBool}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{(isBool } e\texttt{)} : \mathsf{Bool}} \qquad \text{TR-IsNum}\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{(isNum } e\texttt{)} : \mathsf{Bool}}$$

$$\text{TR-Less}\ \frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash \texttt{(< } e_1\ e_2\texttt{)} : \mathsf{Bool}} \qquad \text{TR-Greater}\ \frac{\Gamma \vdash e_1 : \mathsf{Num} \qquad \Gamma \vdash e_2 : \mathsf{Num}}{\Gamma \vdash \texttt{(> } e_1\ e_2\texttt{)} : \mathsf{Bool}}$$

$$\text{TR-Equals}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{(== } e_1\ e_2\texttt{)} : \mathsf{Bool}}$$

$$\text{TR-Id}\ \frac{\Gamma[x] = \tau}{\Gamma \vdash x : \tau} \qquad \text{TR-Let-One}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad (x, \tau_1) :: \Gamma \vdash e_b \cdots : \tau}{\Gamma \vdash \texttt{(let ((}x\ e_1\texttt{))} \ e_b \cdots\texttt{)} : \tau}$$

$$\text{TR-Let-Bindings}\ \frac{\Gamma \vdash e_1 : \tau_1 \qquad (x_1, \tau_1) :: \Gamma \vdash \texttt{(let ((}x_2\ e_2\texttt{)} \cdots\texttt{)} \ e_b\texttt{)} : \tau}{\Gamma \vdash \texttt{(let ((}x_1\ e_1\texttt{) (}x_2\ e_2\texttt{)} \cdots\texttt{)} \ e_b \cdots\texttt{)} : \tau}$$

$$\text{TR-Sequence}\ \frac{\Gamma \vdash e_1 : \tau_1 \cdots \Gamma \vdash e_n : \tau_n}{\Gamma \vdash e_1 \cdots e_n : \tau_n} \qquad \text{TR-Set}\ \frac{\Gamma \vdash e : \tau \qquad \Gamma[x] = \tau}{\Gamma \vdash \texttt{(set } x\ e\texttt{)} : \tau}$$

$$\text{TR-If}\ \frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{(if } e_1\ e_2\ e_3\texttt{)} : \tau} \qquad \text{TR-While}\ \frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 \cdots : \tau}{\Gamma \vdash \texttt{(while } e_1\ e_2 \cdots\texttt{)} : \mathsf{Bool}}$$

Figure 1: Typing rules for Copperhead

- TR-LetBindings describes type-checking of let expressions with multiple bindings in terms of one binding at a time. Since the first binding is visible in the second, the second in the third, and so on, this rule proceeds one binding at a time. There is a separate rule, TR-Let-One, to handle the case of a single binding.[1]

Extensions

These are optional and not for credit, but are interesting to try and discuss in office hours or with your peers:

1. Modify the compiler to use type information in code generation to dramatically simplify the compilation of `isNum` and `isBool`.
2. Add a new type to the definition of `typ` called `NumOrBool`, and enable booleans as `input`. Write a type rule for each of the following cases that exploits the behavior of `isNum` and `isBool` and your knowledge of control flow to make them type-check safely:

   ```
   (if (isNum input) (< input 1) input)
   ```

   ```
   (if (isBool input) (if input 10 5) (+ input 10))
   ```

   Check that your solution works in general for any identifier that has type `NumOrBool`. Implement a less restrictive version of `if` that allows differing types across the then and else branches.

[1] You may find the implied binding-at-a-time matching useful as a suggestion of an implementation strategy.