UCSD CSE131 F19 – Diamondback

October 24, 2019

Due Date: 11pm Wednesday, November 6 **Closed to Collaboration**

You will implement Diamondback, a language with functions and a static type system.

Classroom: `FILL` Github: `FILL`

Syntax

The concrete syntax and type language for Diamondback is below. We use $\cdots$ to indicate *one or more* of the previous element. So a program p is a sequence of one or more definitions d followed by an expression e, and a definition has one or more arguments $x\!:\!\tau$ and one or more expressions in its body $e\cdots$. There are $\boxed{\text{boxes}}$ around the new pieces of concrete syntax.

$$\begin{array}{lcl}
e & := & n \mid \texttt{true} \mid \texttt{false} \mid x \\
 & \mid & \texttt{(let ((}x\ e\texttt{)}\ \cdots\texttt{)}\ e\cdots\texttt{)} \\
 & \mid & \texttt{(if}\ e\ e\ e\texttt{)} \\
 & \mid & \texttt{(}op_2\ e\ e\texttt{)} \mid \texttt{(}op_1\ e\texttt{)} \\
 & \mid & \texttt{(while}\ e\ e\cdots\texttt{)} \mid \texttt{(set}\ x\ e\texttt{)} \\
 & \mid & \boxed{\texttt{(}f\ e\cdots\texttt{)}} \mid \boxed{\texttt{(}f\texttt{)}} \\
d & := & \boxed{\texttt{(def}\ f\ \texttt{(}x\texttt{:}\tau\cdots\texttt{):}\tau\ e\cdots\texttt{)}} \\
 & \mid & \boxed{\texttt{(def}\ f\ \texttt{():}\tau\ e\cdots\texttt{)}} \\
p & := & \boxed{d\cdots e} \\
op_1 & := & \texttt{add1} \mid \texttt{sub1} \mid \texttt{isNum} \mid \texttt{isBool} \mid \boxed{\texttt{print}} \\
op_2 & := & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{<} \mid \texttt{>} \mid \texttt{==} \\
n & := & \text{63-bit signed number literals} \\
x, f & := & \text{variable and function names}
\end{array}$$

$$\begin{array}{lcl}
\tau & := & \texttt{Num} \mid \texttt{Bool} \\
\Delta & := & \{f : \tau\cdots \to \tau, \cdots\} \\
\Delta[f] & \textit{means} & \textit{look up the type of } f \textit{ in } \Delta \\
\Gamma & := & \{x : \tau, \cdots\} \\
\Gamma[x] & \textit{means} & \textit{look up the type of } x \textit{ in } \Gamma \\
(x, \tau) :: \Gamma & \textit{means} & \textit{add } x \textit{ to } \Gamma \textit{ with type } \tau \\
\Delta; \Gamma \vdash e : \tau & \textit{means} & \textit{with definitions } \Delta \textit{ and env } \Gamma, e \textit{ has type } \tau \\
\Delta \vdash_d d : \checkmark & \textit{means} & \textit{with definitions } \Delta \textit{ the definition } d \textit{ type-checks} \\
\vdash_p p : \checkmark & \textit{means} & \textit{the program } p \textit{ type checks}
\end{array}$$

There are two different `def` and function application forms because functions are allowed to have zero arguments, but the $\cdots$ notation means one or more. You can easily represent arguments in both cases with a list that's allowed to be empty.

An example program that computes whether a number is even or odd extremely inefficiently and prints several examples is:

```
(def even (n : Num) : Bool
  (if (== n 0) true (odd (- n 1))))
(def odd (n : Num) : Bool
  (if (== n 0) false (even (- n 1))))
(def test() : Bool
  (print (even 30))
  (print (odd 30))
  (print (even 57))
  (print (odd 57)))

(test)
```

```
Expected output:
true
false
false
true
true
```

Semantics

Function Definitions and Applications

The main new feature in Diamondback is function definitions d and function applications `(`f $e\cdots$`)`. A function application `(`f $e_a\cdots$`)` uses the function definition with the matching name `(def` f `(`x `:`$\tau\cdots$`)` $e\cdots$`)`, and should evaluate to the same result as `(let ((`x e_b`) ...)` $e\cdots$`)`, where the argument expressions $e_a\cdots$ come from the application, the names $x\cdots$ come from the definition, and the body expressions $e_b\cdots$ come from the definition.[1][2]

The Main Expression

Diamondback programs are expected to have a single expression after the definition list that is the main entry point for the program. This expression has `input` bound to the user's input by default.

As an example, consider this program:

```
(def (abs x : Num) : Num
  (if (< x 0) (* -1 x) x))
(* (abs input) 2)
```

We could think of its evaluation taking these steps (with an input of 5):

```
→ (* (abs input) 2)
→ (* (abs 5) 2)
→ (* (let ((x 5)) (if (< x 0) (* -1 x) x)) 2)
→ (* (if (< 5 0) (* -1 5) 5) 2)
→ (* (if false (* -1 5) 5) 2)
→ (* 5 2)
→ 10
```

Printing

Diamondback also adds a new primitive, `print`, that prints a value to the console followed by a newline. Numbers should print as their user-interpreted value, so `(print 22)` should print 22, and `(print true)` should print `true`.

The entire `print` expression evaluates to the same value as its argument (which is the value that gets printed), so `(print (+ 1 7))` evaluates to `8` in addition to printing it.

Type Checking

Diamondback has essentially the same type rules as Copperhead for expressions, with two changes. First, all of the rules contain a definitions environment Δ in addition to the type environment Γ. Second, there are two new rules:

$$\textsc{TR-Print}\ \frac{\Gamma \vdash e : \tau}{\Delta;\Gamma \vdash \texttt{(print } e\texttt{)} : \tau} \qquad \textsc{TR-App}\ \frac{\Delta[f] = \tau_1\cdots\tau_n \to \tau_r \qquad (\Delta;\Gamma \vdash e_1 : \tau_1)\cdots(\Delta;\Gamma \vdash e_n : \tau_n)}{\Delta;\Gamma \vdash \texttt{(}f\ \ e_1\cdots e_n\texttt{)} : \tau_r}$$

[1] Note that replacing application expressions with let expressions is *not* a strategy that works in general in the compiler, because the body expressions $e_b\cdots$ could contain other function applications. This description is, however, a useful way to describe the behavior of a function application succinctly and is a perfectly valid way to evaluate functions "by hand" when we can write out all the intermediate steps with concrete values. See the reading for more detail: `https://ucsd-cse131-f19.github.io/lectures/10-22-lec8/notes.pdf`

[2] We actually need to be a little bit careful here. While this rule worked fine for the single-argument functions in the reading, here we'd really want a version of let that doesn't include earlier bindings in later ones to avoid clashes between names in scope and names in the function's argument list. This is really only relevant if you're writing things out on paper, and doesn't affect the design of a calling convention at all.

TR-Print says that `print` expressions' result is the same type τ as the argument, which matches the semantics.

In English, TR-App rule says

> If the function definition f has argument types $\tau_1 \cdots$ and return type τ_r, and the arguments $e_1 \cdots$ of an application of f have matching types, then the application has type τ_r.

It's common to write the type of definitions as $\tau_1 \cdots \rightarrow \tau_r$, also called an "arrow type", to be evocative of taking a number of argument types and producing a result type.

The existing rules are unchanged aside from tracking Δ (which only TR-App uses).

There are also two new rules, one for type-checking definitions, and one for type-checking programs. They use slightly different $\vdash$ notation with subscripts $_d$ and $_p$ to indicate that they have meaning for pieces of syntax other than expressions. Since, unlike expressions, we don't calculate their overall type, we simply use ✓ in the rule to indicate that they pass all checks.

$$\textsc{TR-Def}\ \frac{\Delta; \{x_1 : \tau_1, \cdots x_n : \tau_n\} \vdash e \cdots : \tau_r}{\Delta \vdash_d \texttt{(def}\ f\ \ \texttt{(}x_1\texttt{:}\tau_1 \cdots x_n\texttt{:}\tau_n\texttt{)}\texttt{:}\tau_r\ \ e \cdots\texttt{)} : \checkmark}$$

$$\textsc{TR-Prog}\ \frac{(\Delta; \vdash_d d_1 : \checkmark) \cdots (\Delta; \vdash_d d_n : \checkmark) \qquad \Delta; \{\texttt{input} : \texttt{Num}\} \vdash e : \tau}{\vdash_p d_1 \cdots d_n e : \checkmark}$$

Where in $\vdash_p$, Δ is constructed by mapping each definition name to an arrow type made of its argument types and return type, so $\texttt{(def}\ f\ \ \texttt{(}x\texttt{:}\tau_1 \cdots x_n\texttt{:}\tau_n\texttt{)}\texttt{:}\tau_r\ \ e \cdots\texttt{)}$ would appear in Δ as $\{f : \tau_1 \cdots \tau_n \rightarrow \tau_r\}$. In English, TR-Def says that a definition type-checks if its body has the expected return type τ_r when type-checked in an environment with just the arguments of the definition mapped to their declared types. A program p type-checks if all of its definitions type-check and its main expression has some type in the environment that assumes `input` has type `Num` (along with also assuming the declared definitions).

New Errors & Miscellaneous

Type errors should be reported with `"Type mismatch"` as usual, including type errors resulting from the new rules, including passing the wrong number of arguments to a function. If a function application uses a function name that isn't defined, the compiler should report an error containing `"Unbound"`.

It's allowed for variables and functions to use the same name, so there could be a top-level definition named `f` and a variable in an argument or let named `f`.

It's a well-formedness error for multiple functions to have the same name, or for multiple arguments within the same function to have the same name. Report these cases with an error that contains the string `"Multiple functions"` and `"Multiple bindings"` respectively.

An empty function body should be reported with `"Invalid"` as with other syntax errors.

Implementation Recommendations and Details

Registers Used by `main`

You may have reasons to want to use registers like `rbx, rbp, rdi` or others. The assembly generated by gcc and clang for `main` may use these registers as well, so they should be saved at the beginning of `our_code_starts_here` and restored before the final `ret`. You can use `push rbx` and `pop rbx` to accomplish this.

Stack Alignment

Some systems require that the stack pointer `rsp` be aligned at a 16-byte (2 word) boundary before making calls into library functions that use system calls, like `printf`. If you get stack alignment segmentation faults[3] you may want to make sure your calling convention always moves `rsp` by multiples of 16, which could mean leaving an extra word of space on some calls. For example, if your calling convention uses 2 words for the old value of `rsp` and the return address, then a function call with an *odd* number of arguments could end up on an 8-byte boundary. You can test this by using `print` in functions with varying numbers of arguments.

Moving Labels into Memory

In class, we used code like `mov [rsp-16], after_call` to move a label into memory. This actually requires two instructions on some platforms. If you see an error like `"format does not support 32-bit absolute addresses."` you may be running into this. The solution is simple, just save the label into a register first:

```
mov rax, after_call
mov [rsp-16], rax
```

Print

While you're free to implement `print` in any way you prefer that works, one that we found expedient is to call a function defined in `main.c`. This requires using C's calling convention. On x86-64, this means moving the argument into register `rdi`, moving `rsp` to free space at the top of the stack, and then using the `call` instruction to push the current code address to the stack and jump to the function you wrote in `main`. On return, `rsp` should be moved back to its original location, and your generated code should ensure that the printed value ends up in `rax`.

Keep in mind that if you use registers other than `rax`, they may be overwritten during the use of print. Wikipedia has a reasonable, brief, accurate summary of the callee-save vs. caller-save behavior at `https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI`. We assuming that users of Diamondback will use `gcc` and `clang` on systems that use this convention, not the Microsoft convention.

Describing Your Calling Convention

As you implement Diamondback, you will need to make a number of decisions, not least of which is the calling convention you choose, and decisions you make around compiling application expressions and definitions. Along with your code, you will write a desgin document as a separate PDF describing how your calling convention works. You should make sure to cover (in whatever order makes sense):

1. A description of your calling convention in general terms:
   (a) What is the caller responsible for vs the callee?
   (b) Do you have to do any particularly interesting work to manage the stack or temporary storage?
   (c) Are there improvements you can imagine making in the future?
2. Pick three example programs that use functions and are interesting in different ways, and use them to describe your calling convention:
   (a) Show their source, generated assembly, and output (you can summarize the generated code if it's quite long)
   (b) Highlight the parts of the generated assembly that make the example interesting, distinct, and/or especially challenging to compile

Still write this even if you don't think you have everything working! In that case, in part 2, pick at least one example that *doesn't* work, and note both its expected output and its actual behavior with your compiler.

[3] We saw this in class `https://github.com/ucsd-cse131-f19/ucsd-cse131-f19.github.io/blob/master/lectures/10-10-lec5/compile.ml#L106`