

Handout 7: Compilation Issues (Revised)

In the last handout, we have looked at abstract machines for executing programs in applied lambda calculi. These machines were capable of taking lambda calculus terms as their programs and compiling them “on the fly.” However, in the normal usage, programs are compiled into machine code ahead of time and the machines are then responsible for only the execution. We examine what issues arise in separating compilation and execution phases of interpreting programs.

1. Multiple argument functions

Recall that in lambda calculi, all functions are unary, i.e., they take a single argument. We have seen that we can simulate multiple-argument functions using unary functions. In this handout, we treat multiple-argument functions directly because it is instructive to look at their implementation issues. The notation for constructing a multiple-argument function is:

$$\lambda(x_1, \dots, x_k). M$$

where x_1, \dots, x_k are distinct variable names. A multiple-argument function M is called by applying it the correct number of arguments:

$$M(N_1, \dots, N_k)$$

If the number of arguments supplied is different from the number of arguments that M expects, we have an error. We will assume that functions are always applied to the correct number of arguments.

We often abbreviate sequences of variables x_1, \dots, x_k as \vec{x} to simplify notation. Similarly, a sequence of terms N_1, \dots, N_k may be abbreviated as \vec{N} .

The SECD machine of Handout 6 can be modified to handle multiple-argument functions by adding an integer parameter k to the `app` instructions. The parameter indicates to the machine how many arguments are present in the function application.

2. Types for multiple-argument functions

Consider a multiple-argument function term $\lambda(x_1, \dots, x_n). M$. If x_1, \dots, x_n are expected to be of types A_1, \dots, A_n and the result is of type B , then the type of the function is written as $A_1 \times \dots \times A_n \rightarrow B$. The symbol “ \times ” is read as “times”, but the mathematical term for the construction is “cross product”.

In contrast, the standard lambda calculus term $\lambda x_1. \dots \lambda x_n. M$ has the type $A_1 \rightarrow \dots \rightarrow A_n \rightarrow B$, which if fully parenthesised, means $A_1 \rightarrow (\dots (A_n \rightarrow B) \dots)$.

For example, the term

$$\lambda x. \lambda y. \text{sqrt}(x * x + y * y)$$

has the type **float** \rightarrow **float** \rightarrow **float**, whereas the term

$$\lambda(x, y). \text{sqrt}(x * x + y * y)$$

has the type **float** \times **float** \rightarrow **float**. The first kind of functions are called *curried functions*, after Haskell Curry who popularised the style. The second type of functions are called *uncurried functions*. There is a curried function corresponding to each multiple-argument function and *vice versa*. So, The two types **float** \rightarrow **float** \rightarrow **float** and **float** \times **float** \rightarrow **float** are in one-to-one correspondence, but they are *different* types.

In programming language, the curried function style is preferred, whereas, in Scheme and ML, the uncurried function style is the preferred style. In this Handout, we use uncurried multiple-argument functions in preference.

We assume that the primitive operators $+$, $*$ etc are multiple-argument functions.

3. Low-level SECD machine

In the SECD machine of Handout 6, all lambda calculus terms are treated as “instructions”. For constants and variables, this is reasonable, because their execution involves only a single transition step. Application terms are not instructions *per se*. The SECD machine of Handout 6 *translates* them into instructions by breaking them up into individual instructions. We are now interested in doing such translation in *advance* so that the SECD machine does not have to do any translation. In other words, we want to split the evaluation of lambda terms into a *compilation* phase and an *execution* phase.

The execution is to be carried out by a low-level SECD machine, whose instructions will have a simpler structure. Using the letter I to stand for individual instructions and C for sequences of instructions, these structures can be described as follows:

$$\begin{aligned} C & ::= [I_1, \dots, I_n] \\ I & ::= c \mid x \mid \text{var}(l, i) \mid \text{app}(k) \mid \text{rtn} \mid \lambda(\vec{x}).C \mid \text{sel}(C_1, C_2) \mid \text{delay}(C) \end{aligned}$$

The forms c for constants and x for variables are the same as before. The form $\text{var}(l, i)$ is used for low-level addressing for variables, which is discussed below. The form $\text{app}(k)$ is the function application instruction for k -argument functions. The form $\lambda(\vec{x}).C$ is obtained by translating lambda abstraction terms. It represents a function with arguments \vec{x} and the body represented by the instructions C . The forms $\text{sel}(C_1, C_2)$ and $\text{delay}(C)$ are similar to the corresponding forms in the high-level SECD machine, except that their constituents are instruction sequences rather than terms. Note that application terms are not instructions. They are expected to be compiled away into lower-level instruction sequences.

One can easily imagine a compiler that goes through a lambda term and translates it into machine language programs of this form. For example, the lambda term $((\lambda(f). \lambda(y). f y) (\lambda(x). +(x, y)) 1)$ would be translated into the following program for call-by-value execution:

$$\begin{aligned} & [\lambda(f). [\lambda(y). [f, y, \text{app}(1)]]], \\ & \lambda(x). [+ , x, y, \text{app}(2)], \text{app}(1), \\ & 1, \text{app}(1) \end{aligned}$$

4. Environment frames and static links

It is wasteful to search for variable names in environments at the execution time. Normally, compilers allocate memory locations to variables, and these memory locations are directly accessed during the execution phase without having to search for variable names. To enable such compilation, we must set things up so that the position of each variable's binding in the environment is fixed, or at least predictable based on the program structure.

To enable such predictability, environments are structured as a linked list of *environment frames*, where each frame contains the bindings made in a particular binding event such as the application of a lambda function. For example, if we evaluate the term $((\lambda(f). \lambda(y). f y) (\lambda(x). +(x, y)) 1)$ in an initial environment $\{y \mapsto 4, z \mapsto 5\}$, the initial bindings are put into a frame $F_0 = \{y \mapsto 4, z \mapsto 5\}$. When the outer level lambda function is applied to $\lambda(x). +(x, y)$, a new frame is created for the binding of f and linked to F_0 . Subsequent application of the second level lambda function to 1 creates a further frame for the binding $y \mapsto 1$. These frames and environments can be depicted as follows:

$$\begin{aligned} E_0 & = F_0 = \{y \mapsto 4, z \mapsto 5\} \\ E_1 & = F_0 \leftarrow \{f \mapsto \text{cls}(\lambda(x). +(x, y), E_0)\} \\ E_2 & = F_0 \leftarrow \{f \mapsto \text{cls}(\lambda(x). +(x, y), E_0)\} \leftarrow \{y \mapsto 1\} \end{aligned}$$

The \leftarrow symbol used in the representation of environments depicts "links" between frames that are actually stored in memory. They are called *static links* because they correspond to the static nesting of scopes in the program structure. The entire environment is a linked list with the rightmost frame being the head of the linked list and the subsequent frames accessible by following the static links. The linked list structure of the environment is referred to as the *static chain*.

The structure of the static chain as well as the variables bound in each of frame can be predicted from the program structure. When evaluating the body of the function $\lambda(f). [\lambda(y). [\dots]]$ the first frame corresponds to the innermost scope, which has a binding for y , the next frame corresponds to the scope of the outer λ function, which has a binding for f , and the last frame contains the bindings for any free variables.

Note that each frame contains at most one binding for any given variable. However, there can be different bindings for the same variable in different frames. For example, there is a binding for y in F_0 and a different binding for y in the last frame $\{y \mapsto 1\}$. When the binding of a variable is looked up, it is searched for sequentially in the static chain. So, the binding for it in the innermost scope is picked up and any bindings in the outer scopes are ignored.

The APPLY rule can now be modified to make use of static chains as environments:

APPLY	$([S \mid \text{cls}(\lambda(\vec{x}). C' E'), \vec{V}], E, [\text{app}(k) \mid C], D)$
\longrightarrow	$(S, E' \leftarrow \{x_1 \mapsto V_1, \dots, x_k \mapsto V_k\}, C' + [\text{rtn} \mid C], [D \mid E])$
	where $\vec{x} = x_1, \dots, x_k$ and $\vec{V} = V_1, \dots, V_k$

Instead of updating the environment as in the original rule, this version adds a new frame to the static chain representing the environment. The symbol $+$ used in the control part denotes concatenation of lists.

5. Translating variables to memory addresses

Let us consider how to translate variable names into memory addresses so that environment look-up is not needed at the execution time. We assume that the environment is structured as a static chain. As mentioned earlier, the structure of the static chain is entirely determined by the program structure. So, the compiler knows, at each point in the program, how many frames would be present in the current environment and what variables would be bound in each of those frames.

Let us assume that each variable binding takes one memory location. (This is an oversimplification, but a real compiler can calculate how much memory is needed for each binding and use that information.) To specify the memory address at which a particular variable's value is to be found, we need to give the level of the frame (the most recent frame being level 0, the next frame being level 1 and so on), and the relative address within that frame for the variable's value. We write this as an instruction $\text{var}(l, i)$ where l is the level of the frame and i is the relative address of the value within the frame (called the "offset"). We show this below for the translation of the expression $((\lambda(f). \lambda(y). f\ y) (\lambda(x). +(x, y))\ 1)$. Both the version with the variable names and that with memory address are shown so that you can see their correspondence:

$$\begin{array}{ll} [\lambda(f). [\lambda(y). [f, y, \text{app}(1)]], & [\lambda(f). [\lambda(y). [\text{var}(1, 0), \text{var}(0, 0), \text{app}(1)]], \\ \lambda(x). [+ , x, y, \text{app}(2)], \text{app}(1), & \lambda(x). [+ , \text{var}(0, 0), \text{var}(1, 0), \text{app}(2)], \text{app}(1), \\ 1, \text{app}(1)] & 1, \text{app}(1)] \end{array}$$

The variable f is bound at one level out from the inner scope and it is located at address 0 within that frame. So, it is translated to $\text{var}(1, 0)$. The location y is bound at the innermost scope at address 0; so it is $\text{var}(0, 0)$. So on.

The rule to load the values of variables specified by such addresses is as given follows:

$\begin{array}{l} \text{LOAD} \quad (S, \quad F_1 \leftarrow \dots \leftarrow F_k, \quad [\text{var}(l, i) \mid C], \quad D) \\ \longrightarrow \quad ([S \mid V], \quad F_1 \leftarrow \dots \leftarrow F_k, \quad C, \quad D) \\ \text{where } V \text{ is the } i\text{'th value stored in } F_{k-l} \end{array}$
--

Note that the level part of the address l is used to count frames from the right. Level 0 accesses F_k , level 1 accesses F_{k-1} and so on.

6. Example To see how the low-level SECD machine works with frames and addresses, examine the following execution trace of our running example. We use the abbreviations:

$$\begin{array}{ll} C_0 \equiv [\text{var}(1, 0), \text{var}(0, 0), \text{app}(1)] & \text{corresponds to } f(y) \\ C_1 \equiv [+ , \text{var}(0, 0), \text{var}(1, 0), \text{app}(2)] & \text{corresponds to } +(x, y) \end{array}$$

	<u>stack</u>	<u>env</u>	<u>control</u>	<u>dump</u>
		F_0	$\lambda(f).[\lambda(y).C_0], \lambda(x).C_1, \text{app}(1), 1, \text{app}(1)$	$[]$
2.	$\text{cls}(\lambda(f).[\lambda(y).C_0], F_0), \text{cls}(\lambda(x).C_1, F_0)$	F_0	$\text{app}(1), 1, \text{app}(1)$	$[]$
3.		$F_0 \leftarrow F_1$	$\lambda(y).C_0, \text{rtn}, 1, \text{app}(1)$	$[E_0]$
6.	$\text{cls}(\lambda(y).C_0, E_1), 1$	F_0	$\text{app}(1)$	$[]$
7.		$F_0 \leftarrow F_1 \leftarrow F_2$	$\text{var}(1, 0), \text{var}(0, 0), \text{app}(1), \text{rtn}$	$[E_0]$
9.	$\text{cls}(\lambda(x).C_1, F_0), 1$	$F_0 \leftarrow F_1 \leftarrow F_2$	$\text{app}(1), \text{rtn}$	$[E_0]$
10.		$F_0 \leftarrow F_3$	$+ , \text{var}(0, 0), \text{var}(1, 0), \text{app}(2), \text{rtn}, \text{rtn}$	$[E_0, E_2]$
13.	$+ , 1, 4$	$F_0 \leftarrow F_3$	$\text{app}(2), \text{rtn}, \text{rtn}$	$[E_0, E_2]$
14.	5	$F_0 \leftarrow F_3$	rtn, rtn	$[E_0, E_2]$
16.	5	F_0		$[]$

The environment frames used in the trace are:

$$\begin{array}{ll} F_0 = \{y \mapsto 4, z \mapsto 5\} & E_0 = F_0 \\ F_1 = \{f \mapsto \text{cls}(\lambda(x).C_1, F_0)\} & E_1 = F_0 \leftarrow F_1 \\ F_2 = \{y \mapsto 1\} & E_2 = F_0 \leftarrow F_1 \leftarrow F_2 \\ F_3 = \{x \mapsto 1\} & E_3 = F_0 \leftarrow F_3 \end{array}$$

The interesting parts to examine are how the APPLY rule works in configurations 2, 6 and 9, building and linking new frames into the static chain environment. Also worth noting is how the environment has the right values at the right place for the execution of the code C_2 (cf. configuration 10), which simulates the expression $+(x, y)$.

7. Java example Similar use of environment frames is made in all programming languages that have variable bindings in nested scopes. For example, here is a Java class that prints messages at periodic intervals to serve as a clock:

```
class PrintingClock {

    public static void start(int interval, boolean beep) {

        ActionListener ticker = new ActionListener() {
            public void actionPerformed(ActionEvent event) {
                Date now = new Date();
                System.out.println(now);
                if (beep) Toolkit.beep();
            }
        };

        Timer t = new Timer(interval, ticker);
        t.start();
    }
}
```

The following frames arise in executing the `start` method of the `PrintingClock`:

$$F_0 = \{\text{Timer} \mapsto \dots, \text{Toolkit} \mapsto \dots, \text{Date} \mapsto \dots, \dots \text{PrintingClock} \mapsto \dots\}$$

$$F_1 = \{\text{start} \mapsto \dots\}$$

$$F_2 = \{\text{interval} \mapsto \dots, \text{beep} \mapsto \dots\}$$

$$F_3 = \{\text{ticker} \mapsto \dots, \text{t} \mapsto \dots\}$$

$$F_4 = \{\text{actionPerformed} \mapsto \dots\}$$

The frame F_0 contains all the names bound at the top-level, mostly class names. A binding for `PrintingClock` is added to this frame. The frame F_1 corresponds to the bindings in the `PrintingClock` class. The frame F_2 is built for binding the parameters of `start` when the method is called. The execution of the method body creates the frame F_3 with bindings for the local variables of the method. While creating the `ticker` object the frame F_4 is built, but nothing is evaluated in this frame.

At periodic intervals, the Java run time system calls the `ticker`'s `actionPerformed` method. Further frames are built at this time:

$$F_5 = \{\text{event} \mapsto \dots\}$$

$$F_6 = \{\text{now} \mapsto \dots\}$$

The final environment in which the code of `actionPerformed` method runs is $F_0 \leftarrow F_1 \leftarrow \dots \leftarrow F_5 \leftarrow F_6$. Note that the binding of the variable `beep` is found at 4 levels out from the inner scope. The method `Toolkit.beep` on the other hand is found by getting the binding of `Toolkit` and looking up inside its value (which would be a structure containing the bindings of method names).