# Hoisting, Allocation and Code Generation

Di Zhao

zhaodi01@mail.ustc.edu.cn

This is the third assignment of Advanced Topics in Software Technologies. In the previous assignment, we implemented closure conversion. After this phase of compilation, all functions are now closed. However, in order to generate low-level target language (such as C programming language) that works properly, a few more steps are needed.

In this lab, there are basically three tasks. First we will perform the procedure named *hoisting*, which lifts all function definitions to top-level. Also, a rough version of allocation (for tuples and tagged values) is included. Secondly, to make the allocation strategy more suitable for garbage collection, we will implement another stage for allocation separately. Finally, the code generation procedure will output the abstract syntax as C code.

After these complements, your Monkey compiler will now be able to compile the ML target language into an executive file.

## 1 Hoisting

The first task in this lab is to hoist all the function definitions to top-level. In this way, a function definition is visible in the entire program. Moreover, the bindings in (nested) "let" expressions are flattened into a list of bindings.

Recall the syntax of closure passing language (the target language in the previous lab) in Figure 1. This will be the source language in this section.

The target language in this section is called Flat. The syntax is shown in Figure 2.

In the Flag language, a program $p$ consists of a main function (denoted as $m$) and a list of functions ($\vec{f}$).

In a function definition $x\ \vec{y}\ \{\ \vec{b};\ \ e;\ \}$, $x$ is the function name, $\vec{y}$ is the list of arguments. The function body consists of a list of bindings $\vec{b}$ and one control transferring expression $e$.

A binding binds a value $v$ to a name $x$. Here the values to be named include empty value, constant integers, constant strings, projection operation, tuples, tagged values and primary operations.

There're three kinds of control transferring expressions (denoted as $e$):

- Function call $x\ \vec{y}$, in which $x$ is the function name and $\vec{y}$ is the arguments.

| | | | |
|---|---|---|---|
| (terms) | $K$ | $\rightarrow$ | $\mathsf{letval}\ x\ =\ V\ \ \mathsf{in}\ \ K$ |
| | | $\mid$ | $\mathsf{let}\ x = \pi_i\ y\ \mathsf{in}\ K$ |
| | | $\mid$ | $\mathsf{letcont}\ k\ env\ x = K\ \mathsf{in}\ K'$ |
| | | $\mid$ | $k\ env\ x$ |
| | | $\mid$ | $f\ env\ k\ x$ |
| | | $\mid$ | $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\ x_1 \Rightarrow K \mid \mathsf{in}_2\ x_2 \Rightarrow K'$ |
| | | $\mid$ | $\mathsf{letprim}\ x = PrimOp\ \vec{y}\ \ \mathtt{in}\ K$ |
| | | $\mid$ | $\mathsf{if0}\ x\ \mathsf{then}\ K\ \mathsf{else}\ K'$ |
| | | $\mid$ | $\mathsf{letfix}\ f\ env\ k\ x = K\ \mathsf{in}\ K'$ |
| (values) | $V$ | $\rightarrow$ | $()$ |
| | | $\mid$ | $i$ |
| | | $\mid$ | "$s$" |
| | | $\mid$ | $(x_1, x_2,\ \ldots\ , x_n)$ |
| | | $\mid$ | $\mathsf{in}_i\ x$ |
| | | $\mid$ | $\lambda env\ k\ x.K$ |
| (primitives) | $PrimOp$ | $\rightarrow$ | $+$ |
| | | $\mid$ | $-$ |
| | | $\mid$ | $*$ |
| | | $\mid$ | $\mathsf{print}$ |
| | | $\mid$ | $\mathsf{int2string}$ |

Figure 1: Closure syntax

| | | | |
|---|---|---|---|
| (program) | $p$ | $\rightarrow$ | $m;\ \vec{f}$ |
| (functions) | $m,\ f$ | $\rightarrow$ | $x\ \vec{y}\ \{\ \vec{b};\ \ e;\ \}$ |
| (bindings) | $b$ | $\rightarrow$ | $x = v$ |
| (values) | $v$ | $\rightarrow$ | $()$ |
| | | $\mid$ | $i$ |
| | | $\mid$ | "$s$" |
| | | $\mid$ | $\pi_i\ x$ |
| | | $\mid$ | $(x_1, x_2,\ \ldots\ , x_n)$ |
| | | $\mid$ | $\mathsf{in}_i\ x$ |
| | | $\mid$ | $PrimOp\ \vec{x}$ |
| (primitives) | $PrimOp$ | $\rightarrow$ | $+$ |
| | | $\mid$ | $-$ |
| | | $\mid$ | $*$ |
| | | $\mid$ | $\mathsf{print}$ |
| | | $\mid$ | $\mathsf{int2string}$ |
| (transfers) | $e$ | $\rightarrow$ | $x\ \vec{y}$ |
| | | $\mid$ | $\mathtt{if0}\ x\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2$ |
| | | $\mid$ | $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\ x_1 \Rightarrow e_1 \mid \mathsf{in}_2\ x_2 \Rightarrow e_2$ |

Figure 2: Flat syntax

- If expression `if0` $x$ `then` $e_1$ `else` $e_2$, in which $e_1$ and $e_2$ are two control transferring expressions in tern.
- Case expression $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\ x_1 \Rightarrow e_1 \mid \mathsf{in}_2\ x_2 \Rightarrow e_2$, in which $e_1$ and $e_2$ are two control transferring expressions in tern.

The hoisting procedure takes a Closure syntax term and recursively generates a triple, whose components are:

1. a list of functions;
2. a list of bindings;
3. one control transferring expression.

The hoisting rules are illustrated in Figure 3.

Each rule in Figure 3 consists of some premises (above the horizontal line) and the conclusion (below the horizontal line). The symbol " $\rightsquigarrow$ " means "leads to".

When hoisting a term in which a function is defined, a new function in Flat syntax is created by recursively hoisting the function body. The bindings and control transferring expression of the "current" function is obtained by hoisting the body of the original term. All the functions generated via hoisting sub-terms are added to the function list together with the newly created function. See rule: H-LetValFunc, H-LetCont, H-LetFix.

When hoisting a term that binds a value (other than an abstraction) to a name, simply hoist its sub-term recursively and then add the binding ahead of the binding list. See rule: H-LetVal, H-Let, H-LetPrim. You may want to ask yourself this question: why the binding is added at the front?

When hoisting a term with control transfer, a corresponding control transferring expression is returned along with the function and binding list generated via recursive hoisting (if any). See rule: H-ContApp, H-FuncApp, H-Case, H-If0.

Notice that a source program $t$ is one single syntax term, it will be translated into a list of bindings $\vec{b}$ and one expression $e$ along with the flattened functions. $\vec{b}$ and $e$ will be packed as the main function of the target program. Consequently, in the Flat syntax, the main function of a program has no arguments.

**Exercise 1**. Finish the function `hoistExp` in file `hoist.sml` for the conversion from Closure.t to the Flat syntax. You may want to read through file `flat.sig` first to see how the Flat syntax is defined.

In `hoist.sml`, we use a global reference of function list (`funcs`) to store the flattened functions. In this way, in `hoistExp` we only need to pass a binding list and a control transferring expression as the result.

$$\frac{\vdash K \rightsquigarrow (\vec{f_1}, \vec{b_1}, e_1) \quad \vdash K' \rightsquigarrow (\vec{f_2}, \vec{b_2}, e_2)}{\vdash \mathsf{letval}\ x = \lambda env\ k\ z.K\ \mathsf{in}\ K' \rightsquigarrow (\vec{f_1} :: \vec{f_2}, :: f_x,\ \vec{b_2},\ e_2)} \text{ (H-LetValFunc)}$$
where $f_x$ is function: $x\ [env, k, z]\{\vec{b_1};\ e_1;\}$

$$\frac{\vdash K \rightsquigarrow (\vec{f}, \vec{b}, e)}{\vdash \mathsf{letval}\ x = V\ \mathsf{in}\ K \rightsquigarrow (\vec{f}, (x = V) :: \vec{b}, e)} \text{ (H-LetVal)}$$

$$\frac{\vdash K \rightsquigarrow (\vec{f}, \vec{b}, e)}{\vdash \mathsf{let}\ x = \pi_i\ y\ \mathsf{in}\ K \rightsquigarrow (\vec{f}, (x = \pi_i\ y) :: \vec{b}, e)} \text{ (H-Let)}$$

$$\frac{\vdash K \rightsquigarrow (\vec{f_1}, \vec{b_1}, e_1) \quad \vdash K' \rightsquigarrow (\vec{f_2}, \vec{b_2}, e_2)}{\vdash \mathsf{lecont}\ k\ env\ x = K\ \mathsf{in}\ K' \rightsquigarrow (\vec{f_1} :: \vec{f_2}, :: f_k,\ \vec{b_2},\ e_2)} \text{ (H-LetCont)}$$
where $f_k$ is function: $k\ [env, x]\{\vec{b_1};\ e_1;\}$

$$\frac{}{\vdash k\ env\ x \rightsquigarrow ([\,], [\,], k\ [env, x])} \text{ (H-ContApp)}$$

$$\frac{}{\vdash f\ env\ k\ x \rightsquigarrow ([\,], [\,], f\ [env, k, x])} \text{ (H-FuncApp)}$$

$$\frac{\vdash K \rightsquigarrow (\vec{f_1}, \vec{b_1}, e_1) \quad \vdash K' \rightsquigarrow (\vec{f_2}, \vec{b_2}, e_2)}{\vdash \mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\ x_1 \Rightarrow K \mid \mathsf{in}_2\ x_2 \Rightarrow K' \rightsquigarrow} \text{ (H-Case)}$$
$$(\vec{f_1} :: \vec{f_2},\ \vec{b_1} :: \vec{b_2},\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\ x_1 \Rightarrow e_1 \mid \mathsf{in}_2\ x_2 \Rightarrow e_2)$$

$$\frac{\vdash K \rightsquigarrow (\vec{f}, \vec{b}, e)}{\vdash \mathsf{letprim}\ x = PrimOp\ \vec{y}\ \texttt{in}\ K \rightsquigarrow (\vec{f}, (x, PrimOp\ \vec{y}) :: \vec{b}, e)} \text{ (H-LetPrim)}$$

$$\frac{\vdash K \rightsquigarrow (\vec{f_1}, \vec{b_1}, e_1) \quad \vdash K' \rightsquigarrow (\vec{f_2}, \vec{b_2}, e_2)}{\vdash \mathsf{if0}\ x\ \mathsf{then}\ K\ \mathsf{else}\ K' \rightsquigarrow} \text{ (H-If0)}$$
$$(\vec{f_1} :: \vec{f_2},\ \vec{b_1} :: \vec{b_2},\ \texttt{if0}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2)$$

$$\frac{\vdash K \rightsquigarrow (\vec{f_1}, \vec{b_1}, e_1) \quad \vdash K' \rightsquigarrow (\vec{f_2},\ \vec{b_2},\ e_2)}{\vdash \mathsf{letfix}\ f\ env\ k\ x = K\ \mathsf{in}\ K' \rightsquigarrow (\vec{f_1} :: \vec{f_2}, :: f_f,\ \vec{b_2},\ e_2)} \text{ (H-LetFix)}$$
where $f_f$ is function: $f\ [env, k, x]\{\vec{b_1};\ e_1;\}$

Figure 3: Hoisting rules for Closure.t

**Exercise 2**. Finish the function `Exp.dump2file` and function `Binding.dump2file` in file `flat.sml` to pretty print a flattened syntax term. As we haven't dealt with types, the output is not likely to pass the compilation of SML/NJ. To check your implementation, it is recommended that you run some test and examine the outcome before moving on.

## 2 Allocation

The second phase we will implement in this lab is Allocation. After this process, the allocation and initialization of tuples and tagged values will be explicit. Besides, the arguments of a single function are packed into one single argument, so that it would be easier to scan over the arguments when performing garbage collection. The target language in this phase is called: Machine. The syntax of Machine language is shown in Figure 4.

| | | | |
|---|---|---|---|
| (program) | $P$ | $\rightarrow$ | $m;\ \vec{f}$ |
| (functions) | $m,\ f$ | $\rightarrow$ | $x\ y\ \{\ \overrightarrow{binding};\ \ b;\ \}$ |
| (bindings) | $binding$ | $\rightarrow$ | $x = v$ |
| | | $\mid$ | $dst\texttt{[}i\texttt{]} = src$ |
| (values) | $v$ | $\rightarrow$ | `null` |
| | | $\mid$ | $i$ |
| | | $\mid$ | "$s$" |
| | | $\mid$ | $x\texttt{[}i\texttt{]}$ |
| | | $\mid$ | `allocTuple`$(i)$ |
| | | $\mid$ | `allocTag`$(i)$ |
| | | $\mid$ | $PrimOp\ \vec{x}$ |
| (primitives) | $PrimOp$ | $\rightarrow$ | $+$ |
| | | $\mid$ | $-$ |
| | | $\mid$ | $*$ |
| | | $\mid$ | print |
| | | $\mid$ | int2string |
| (blocks) | $b$ | $\rightarrow$ | $\overrightarrow{binding};\ \ e$ |
| (transfers) | $e$ | $\rightarrow$ | $x\ y$ |
| | | $\mid$ | `if0` $x$ `then` $b_1$ `else` $b_2$ |
| | | $\mid$ | case $x$ of $\mathsf{in}_1\ x_1 \Rightarrow b_1 \mid \mathsf{in}_2\ x_2 \Rightarrow b_2$ |

Figure 4: Machine syntax

After the allocation procedure, it will be more convenient to perform code generation based on the Machine language.

The syntax of Machine is similar to the Flat syntax. The differences are listed below:

- To facilitate garbage collection, all functions now takes one single argument.
- There are now two categories of bindings (*binding* in Figure 4) in Machine:
  1. normal bindings ($x = v$) that binds a value to a name;

2. initializations ($dst\texttt{[}i\texttt{]} = src$) that initialize the component $i$ of a tuple or tagged value $dst$ with $src$.

- Instead of tuples, now we have `allocTuple`(i), representing the allocation of a tuple with $i$ components.
- Instead of tagged values, now we have `allocTag`(i), representing the allocation of a tagged value with $i$ as the tag.
- In Machine language, we extend the syntax with another kind of term named *block*. A block (denoted as $b$ in Figure 4) consists of a list of bindings and one control transferring expression. A block can serve as a component of a function or control transferring expression. We introduce the block syntax term for the reason that, as in Machine there's only one argument in a function call, we have to attach some bindings ahead to pack the original arguments in a tuple. You'll see this more explicitly in Figure 6.

Given the Machine syntax, we can decide the conversion rules intuitively, as shown in Figure 5, 6, 7.

$$
\begin{aligned}
\mathcal{A}_b : \text{string} * \text{Flat.binding} &\rightarrow \text{Machine.binding list} \\
\mathcal{A}_b(x = ()) &= [x = \texttt{null}] \\
\mathcal{A}_b(x = i) &= [x = i] \\
\mathcal{A}_b(x = \text{"}s\text{"}) &= [x = \text{"}s\text{"}] \\
\mathcal{A}_b(x = \pi_i\ y) &= [x = y\texttt{[}i\texttt{]}] \\
\mathcal{A}_b(x = (x_1, x_2,\ \ldots\ , x_n)) &= [x = \texttt{allocTuple}(n), \\
&\qquad x\texttt{[1]} = x_1, \\
&\qquad \ldots \\
&\qquad x\texttt{[}n\texttt{]} = x_n] \\
\mathcal{A}_b(x = \mathsf{in}_i\ y) &= [x = \texttt{allocTag}(i), \\
&\qquad x\texttt{[1]} = y] \\
\mathcal{A}_b(x = PrimOp\ \vec{y}) &= [x = PrimOp\ \vec{y}]
\end{aligned}
$$

Figure 5: Conversion from Flat to Machine for bindings

As for bindings, because tuples and tagged values are structured data elements, we need to allocate space for them and perform some initializations. In this way, both of them require special treatment:

- For $x = (x_1, ..., x_n)$, allocate a tuple $x$ with $n$ components and initialize component $x\texttt{[}i\texttt{]}$ with $x_i$.

$$
\begin{aligned}
\mathcal{A}_e : \text{string} * \text{Flat.e} &\rightarrow \text{Machine.block} \\
\mathcal{A}_e(x\ \vec{y}) &= [z = \texttt{allocTuple}(n), \\
&\quad\ z\texttt{[1]} = y_1, \\
&\quad\ \ldots \\
&\quad\ z\texttt{[}n\texttt{]} = y_n]; \\
&\quad\ x\ z \\
&\quad\ (\text{given that } \vec{y} = [y_1,\ \ldots, y_n]) \\
\mathcal{A}_e(\texttt{if0}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2) &= [\,]; \\
&\quad\ \texttt{if0}\ x\ \texttt{then}\ \mathcal{A}_e(e_1)\ \texttt{else}\ \mathcal{A}_e(e_2) \\
\mathcal{A}_e(\mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\ x_1\ \Rightarrow e_1 \mid \mathsf{in}_2\ x_2 \Rightarrow e_2) &= [\,]; \\
&\quad\ \mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\ x_1\ \mathcal{A}_e(e_1) \mid \mathsf{in}_2\ x_2 \Rightarrow \mathcal{A}_e(e_2)
\end{aligned}
$$

Figure 6: Conversion from Flat to Machine for expressions

$$
\begin{aligned}
\mathcal{A}_f(x\ \vec{y}\ \{\ \vec{b};\ e;\}) = x\ (z)\ \{ & \\
&\overrightarrow{y_i = z[i]} :: \\
&\overrightarrow{\mathcal{A}_b(b)}; \\
&\mathcal{A}_e(e); \\
\}& \\
(\text{given that } \vec{y} = [y_1,\ \ldots, y_n],\ i = 1,\ \ldots, n)&
\end{aligned}
$$

Figure 7: Conversion from Flat to Machine for functions

- For $x = in_i\ y$, allocate structure $x$ with tag value $i$ and initialize component $x\texttt{[1]}$ with $y$.

Arguments of a function call now need to be packed into a tuple, and the tuple needs to be initialized by these arguments. In this way, control transferring expressions will be translated into a *block* which consists of a list of bindings and an expression.

For the same reason above, in a function definition, the original arguments need to be fetched out from the sole parameter of the function. Namely we need to add some bindings ahead of the original function body.

Pay attention that as the main function $m$ takes no arguments, no additional bindings for argument folding is needed.

**Exercise 3**. Finish the functions in file `codegen.sml` for the conversion from Flat syntax tree to the Machine syntax tree. You need to finish the three functions below:

- `transExp` to translate a control transferring expression.
- `transOneBinding` to translate a binding.
- `transFunction` to translate a function.

You may want to read through file `machine.sig` first to see how the Machine syntax is defined.

# 3 Code Generation

The last task for you in this lab is to implement the code generation procedure. In this part of the lab, you will encode the functions to output the Machine syntax tree into a `.c` file which can be compiled by a C compiler and then be executed.

As is shown Figure 4, the Machine syntax is basically C style syntax. Consequently we can output C code for most of the syntax terms directly without making much modifications. However, there are a few aspects to which you may pay attention:

- Type and type-casting. As C is strong typed, you'll have to declare variables and insert type-castings with explicit types. For example, you can declare all the variables with `int*` type; when performing a function call, you may cast the function name into `void * (*)()` type first.
- Bindings of tuples and tagged values are encoded as function calls to `allocTuple` and `allocTag` respectively. You'll need to implement these two functions in the runtime system.
- A *case* expression $\mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\ x_1\ \Rightarrow e_1 \mid \mathsf{in}_2\ x_2 \Rightarrow e_2$ can be encoded as an switch structure, whose condition is the tag value $i$ of $x$. Besides, variable $x_1$ and $x_2$ should be initialized (by what value?) before the switch structure.

To support the output code, we need to provide a runtime system, including the definition of function `allocTuple` and `allocTag`. Therefore, we need to decide the memory map for these structures.

As we rely on garbage collection to retrieve memory space, some additional information needs to be attached, such as the length of a tuple, or to distinguish between integers, tuples and tagged values. We will discuss about this with more details in the next lab. For now we just use a simple strategy shown in Figure 8 and 9.

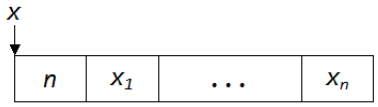


Figure 8: memory map for tuple $x = (x_1,\ ...,\ x_n)$

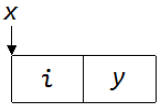


Figure 9: memory map for tagged value $x = \mathsf{in}_i\ y$

As is shown in Figure 8, a tuple with $n$ components will occupy $n+1$ memory cells. The first cell is to store the length of the tuple, the rest $n$ cells store the component values in order. In this lab, each cell is constraint to be 4kb. In this way, `allocTuple(n)` will allocate a memory space of $4*(n+1)$kb and store the value $n$ in the first cell. Finally it returns a pointer to the first cell. The value of the components will be loaded later via this pointer.

Note that we will never visit a tuple with the length of 0, so we can simply output "`x=0`" instead of "`x=allocTuple(0)`".

As shown in Figure 9, a tagged value $\mathsf{in}_i\ y$ takes two memory cells. The first cell is to store the tag value $i$ (1 or 2), while the second one for the value to be tagged. In this way, `allocTag(i)` will allocate a memory space of 8kb and store the tag value $i$ in the first cell. And finally it will return a pointer to the first cell. The value $y$ will be loaded later via this pointer.

**Exercise 4**. Finish function `allocTuple` and `allocTag` in file `test/runtime.c`. You should allocate memory space and do some initializations according to Figure 8 and 9. Finally you should return a pointer to the proper memory address.

We have provided some examples in folder `example` to test your implementation for the runtime system.

**Exercise 5**. Finish the `dump2file'` functions for expressions and bindings, in file `machine.sml`. The `dump2file` functions will output a syntax term as C code.

This completes the code generation procedure. After you have compiled everything (in SML/NJ), you can test your code by taping "`Main.main(t, "fname")`". Where `t` is MLAst.t, and `fname` is the prefix of the output files. If everything is correct, an executive file will be generated (you need to install a C compiler such as gcc, see `Main.sml`).

The examples in folder `example` may offer you some clues.

We have provided some test cases in `test.sml`. Yet you may want to write some of your own examples to test your Monkey compiler.