A Tiger Intermediate Language Specification

May 5, 2009

tree-exp ::= *num*
| *label*
| *temp*
| (*biop tree-exp tree-exp*)
| (`mem` *tree-exp*)
| (`call` *fn tree-exp*)
| (`eseq` *tree-stm tree-exp*)
tree-stm ::= (`move` (`mem` *tree-exp*) *tree-exp*)
| (`move` *temp tree-exp*)
| (`texp` *tree-exp*)
| (`jump` *tree-exp label* ...)
| (`cjump` *relop tree-exp tree-exp label label*)
| (`seq` *tree-stm tree-stm tree-stm* ...)
| *label*
relop ::= *eqop*
| <= | >= | < | >
eqop ::= = | <>
fn ::= “allocate”
| “printstr”
| “printint”
| “printant”

Figure 1: The Tiger intermediate language

pure-exp ::= *num*
| *label*
| *temp*
| (*biop pure-exp pure-exp*)
| (`mem` *pure-exp*)

$$
\begin{aligned}
\mathsf{Eval}[\![S, \mathit{num}]\!] &= \mathit{num} \\
\mathsf{Eval}[\![S, \mathit{label}]\!] &= \mathit{label} \\
\mathsf{Eval}[\![S, \mathit{temp}]\!] &= \mathsf{lookup}[\![S, \mathit{temp}]\!] \\
\mathsf{Eval}[\![S, (\texttt{mem}\ \mathit{pure\text{-}exp})]\!] &= \mathsf{lookup}[\![S, \mathsf{Eval}[\![S, \mathit{pure\text{-}exp}]\!]\ ]\!] \\
\mathsf{Eval}[\![S, (\mathit{biop}\ \mathit{pure\text{-}exp}_1\ \mathit{pure\text{-}exp}_2)]\!] &= \delta[\![\mathit{biop}, \mathsf{Eval}[\![S, \mathit{pure\text{-}exp}_1]\!]\ , \mathsf{Eval}[\![S, \mathit{pure\text{-}exp}_2]\!]\ ]\!]
\end{aligned}
$$

Figure 2: Pure tree expressions

1 Overview

This document describes the intermediate language for the Tiger compiler. It is a language that contains both statements (the *tree-stm* non-terminal) and expressions (the *tree-exp* non-terminal), show in figure 1.

The semantics for the language is given as a rewriting system that rewrites a store plus a sequence of expressions, moving a program counter (represented by `pc`) through the sequence of statements. In general, program evaluation proceeds by advancing the program counter through the series of statements, performing the effects of the statements as they pass by. If a `jump` statement is encountered, the program counter is moved to just after the corresponding label. In order for this to work, however, the statements must have be sanitized so that they do not contain embedded statements, since those embedded statements might have labels that could be the target of a jump. Thus, there are a number of rules that simplify *tree-exp*s into *pure-tree-exp*. Figure 2 contains the definition of *pure-tree-exp*. They are just like *tree-exp*s, except they do not contain statements or function calls. Figure 2 also shows the evaluator for pure expressions.

2 Move rules

Before seeing how arbitrary expressions can be turned into pure expressions, first consider the rules that handle the case where the expressions are already pure. The first of these are the `move` rules, shown in figure 3. The first rule shows what happens when a `move` expression encounters a label and its argument is a pure expression. It advances the program

(*S* *tree-stm*$_{before}$ ... `pc` (`move` *temp pure-exp*$_1$) *tree-stm*$_{after}$ ...)	$\longrightarrow$ (update[[*S*, *temp*, Eval[[*S*, *pure-exp*$_1$]]]] *tree-stm*$_{before}$ ... (`move` *temp pure-exp*$_1$) `pc` *tree-stm*$_{after}$ ...)	[move-temp-exp]
(*S* *tree-stm*$_{before}$ ... `pc` (`move` (`mem` *pure-exp*$_1$) *pure-exp*$_2$) *tree-stm*$_{after}$ ...)	$\longrightarrow$ (update[[*S*, Eval[[*S*, *pure-exp*$_1$]] , Eval[[*S*, *pure-exp*$_2$]]]] *tree-stm*$_{before}$ ... (`move` (`mem` *pure-exp*$_1$) *pure-exp*$_2$) `pc` *tree-stm*$_{after}$ ...)	[move-mem-exp]
(*S* *tree-stm*$_{before}$ ... `pc` (`move` (`mem` *pure-exp*$_1$) (`call` *fn pure-exp*$_2$)) *tree-stm*$_{after}$ ...)	$\longrightarrow$ (*S* *tree-stm*$_{before}$ ... `pc` (`move` `r:temp` (`call` *fn pure-exp*$_2$)) (`move` (`mem` *pure-exp*$_1$) `r:temp`) *tree-stm*$_{after}$ ...) where `r:temp` fresh	[move-mem-call]
(*S* *tree-stm*$_{before}$ ... `pc` (`move` *temp* (`call` "allocate" *pure-exp*)) *tree-stm*$_{after}$ ...)	$\longrightarrow$ (alloc[[*S*, *temp*, Eval[[*S*, *pure-exp*]]]] *tree-stm*$_{before}$ ... (`move` *temp* (`call` "allocate" *pure-exp*)) `pc` *tree-stm*$_{after}$ ...)	[move-temp-alloc]
(*S* *tree-stm*$_{before}$ ... `pc` (`move` *temp* (`call` *fn pure-exp*)) *tree-stm*$_{after}$ ...)	$\longrightarrow$ (update[[*S*, *temp*, 0]] *tree-stm*$_{before}$ ... (`move` *temp* (`call` *fn pure-exp*)) `pc` *tree-stm*$_{after}$ ...) where *fn* ≠ "allocate"	[move-temp-fn]

Figure 3: Move reductions

counter past the `move` expression and then updates the store with the value of the argument to move.

The second rule covers a similar case: when a move expression updates a memory location. The difference between it and the previous rule is that the evaluator must be invoked twice, once on the argument to `mem` (to find the memory location), and once for the value to be saved.

The next two rules cover the case where the `move` expression moves the result of a call to a function. If the result of the function call is to be stored in memory, the [move-mem-call] rule simply rewrites it into a move to a register and the moves the value of the register into the memory location (without advancing the program counter).

The [move-temp-alloc] covers the case where the allocation function is called. It moves the program counter is moved past the allocation and updates the store via the alloc function. Its definition is not shown, but it returns a number that refers to a memory address in the store and initializes the appropriate number of words. Note that allocate's argument is a number of words (not bytes), and it returns a pointer to a space that is initialized (to zero).

The [move-temp-fn] function covers the other builtin functions, but the model does not explicitly cover IO, so they are just skipped.

$$\begin{array}{lll}
\begin{array}{l}(S\\ \quad \textit{tree-stm}_{\textit{before}}\ \ldots\\ \quad \texttt{pc}\\ \quad (\texttt{jump}\ \textit{pure-exp}\ \textit{label}\ \ldots)\\ \quad \textit{tree-stm}_{\textit{after}}\ \ldots)\end{array}
& \longrightarrow \begin{array}{l}\mathsf{MovePC}[\![\mathsf{Eval}[\![S, \textit{pure-exp}]\!]\ ,\\ \quad (S\\ \quad\ \textit{tree-stm}_{\textit{before}}\ \ldots\\ \quad\ (\texttt{jump}\ \textit{pure-exp}\ \textit{label}\ \ldots)\\ \quad\ \textit{tree-stm}_{\textit{after}}\ \ldots)]\!]\end{array}
& [\mathsf{jump}]
\end{array}$$

$$\begin{array}{lll}
\begin{array}{l}(S\\ \quad \textit{tree-stm}_{\textit{before}}\ \ldots\\ \quad \texttt{pc}\\ \quad (\texttt{cjump}\ \textit{biop}\\ \qquad\qquad \textit{pure-exp}_1\ \textit{pure-exp}_2\\ \qquad\qquad \textit{label}_n\ \textit{label}_0)\\ \quad \textit{tree-stm}_{\textit{after}}\ \ldots)\end{array}
& \longrightarrow \begin{array}{l}\mathsf{MovePC}[\![\textit{label}_n,\\ \quad (S\\ \quad\ \textit{tree-stm}_{\textit{before}}\ \ldots\\ \quad\ (\texttt{cjump}\ \textit{biop}\\ \qquad\qquad \textit{pure-exp}_1\ \textit{pure-exp}_2\\ \qquad\qquad \textit{label}_n\ \textit{label}_0)\\ \quad\ \textit{tree-stm}_{\textit{after}}\ \ldots)]\!]\end{array}
& [\mathsf{cjump\text{-}true}]\\
\multicolumn{3}{c}{\text{where } \mathsf{Nonzero?}[\![\mathsf{Eval}[\![S, (\textit{biop}\ \textit{pure-exp}_1\ \textit{pure-exp}_2)]\!]\ ]\!]}
\end{array}$$

$$\begin{array}{lll}
\begin{array}{l}(S\\ \quad \textit{tree-stm}_{\textit{before}}\ \ldots\\ \quad \texttt{pc}\\ \quad (\texttt{cjump}\ \textit{biop}\\ \qquad\qquad \textit{pure-exp}_1\ \textit{pure-exp}_2\\ \qquad\qquad \textit{label}_n\ \textit{label}_0)\\ \quad \textit{tree-stm}_{\textit{after}}\ \ldots)\end{array}
& \longrightarrow \begin{array}{l}\mathsf{MovePC}[\![\textit{label}_0,\\ \quad (S\\ \quad\ \textit{tree-stm}_{\textit{before}}\ \ldots\\ \quad\ (\texttt{cjump}\ \textit{biop}\\ \qquad\qquad \textit{pure-exp}_1\ \textit{pure-exp}_2\\ \qquad\qquad \textit{label}_n\ \textit{label}_0)\\ \quad\ \textit{tree-stm}_{\textit{after}}\ \ldots)]\!]\end{array}
& [\mathsf{cjump\text{-}false}]\\
\multicolumn{3}{c}{\text{where } \mathsf{Zero?}[\![\mathsf{Eval}[\![S, (\textit{biop}\ \textit{pure-exp}_1\ \textit{pure-exp}_2)]\!]\ ]\!]}
\end{array}$$

$$\mathsf{MovePC}[\![\textit{label}, (S\ \textit{tree-stm}_{\textit{before}}\ \ldots\ \textit{label}\ \textit{tree-stm}_{\textit{after}}\ \ldots)]\!]\ =\ (S\ \textit{tree-stm}_{\textit{before}}\ \ldots\ \textit{label}\ \texttt{pc}\ \textit{tree-stm}_{\textit{after}}\ \ldots)$$

Figure 4: Jump reductions

3 Jump rules

The jump rules are shown in figure 4. They hinge on the MovePC function. For [jump], it evaluates the argument to `jump`, and then calls MovePC, supplying the value of `jump`'s argument, as well as the machine state – but without a program counter. Then, the MovePC function simply inserts the program counter right before the target of the jump (as shown in the bottom of the figure).

Similarly, the `cjump` rules evaluate the arguments to `cjump` and then jump to one or the other target (the two side-conditions ensure that only rule fires).

4 Boring rules

The rules in figure 5 simply advance the program counter past labels and pure expressions.

$$\begin{array}{lll}
\begin{array}{l}(S\\ \quad \textit{tree-stm}_{\textit{before}}\ \ldots\\ \quad \texttt{pc}\\ \quad \textit{label}\\ \quad \textit{tree-stm}_{\textit{after}}\ \ldots)\end{array}
& \longrightarrow \begin{array}{l}(S\\ \quad \textit{tree-stm}_{\textit{before}}\ \ldots\\ \quad \textit{label}\\ \quad \texttt{pc}\\ \quad \textit{tree-stm}_{\textit{after}}\ \ldots)\end{array}
& [\mathsf{label}]
\end{array}$$

$$\begin{array}{lll}
\begin{array}{l}(S\\ \quad \textit{tree-stm}_{\textit{before}}\ \ldots\\ \quad \texttt{pc}\\ \quad (\texttt{texp}\ \textit{pure-exp})\\ \quad \textit{tree-stm}_{\textit{after}}\ \ldots)\end{array}
& \longrightarrow \begin{array}{l}(S\\ \quad \textit{tree-stm}_{\textit{before}}\ \ldots\\ \quad (\texttt{texp}\ \textit{pure-exp})\\ \quad \texttt{pc}\\ \quad \textit{tree-stm}_{\textit{after}}\ \ldots)\end{array}
& [\mathsf{texp}]
\end{array}$$

Figure 5: Expression and label reductions

5 Flattening rules

The rules in figure 6 cover the flattening operation. The first flattening rule is straightforward; if the statement following the program counter is a sequence, simply flatten out the sequence. The second and third rules involve the flatten-S and flatten-E contexts. Without looking at those contexts yet, the intuition for these rules is that they simply pull out the first

(*S* *tree-stm*$_{before}$ ... `pc` (`seq` *tree-stm*$_{1}$ ...) *tree-stm*$_{after}$ ...)	⟶ (*S* *tree-stm*$_{before}$ ... `pc` *tree-stm*$_{1}$ ... *tree-stm*$_{after}$ ...)	[flatten-seq]
(*S* *tree-stm*$_{before}$ ... `pc` *flatten-S*[(`eseq` *tree-stm tree-exp*)] *tree-stm*$_{after}$ ...)	⟶ (*S* *tree-stm*$_{before}$ ... `pc` *tree-stm* *flatten-S*[*tree-exp*] *tree-stm*$_{after}$ ...)	[flatten-eseq]
(*S* *tree-stm*$_{before}$ ... `pc` *flatten-S*[*flatten-E1*[(`call` *fn pure-exp*)]] *tree-stm*$_{after}$ ...)	⟶ (*S* *tree-stm*$_{before}$ ... `pc` (`move` `r:temp` (`call` *fn pure-exp*)) *flatten-S*[*flatten-E1*[`r:temp`]] *tree-stm*$_{after}$ ...) where `r:temp` fresh	[flatten-call]

Figure 6: Flattening reductions

flatten-S ::= (`move` (`mem` *flatten-E*) *tree-exp*)
| (`move` (`mem` *pure-exp*) *flatten-E*)
| (`move` *temp flatten-E*)
| (`texp` *flatten-E*)
| (`jump` *flatten-E label* ...)
| (`cjump` *relop flatten-E tree-exp label label*)
| (`cjump` *relop pure-exp flatten-E label label*)
flatten-E ::= []
| *flatten-E1*[*flatten-E*]
flatten-E1 ::= (`eseq` *flatten-S tree-exp*)
| (*biop* [] *tree-exp*)
| (*biop pure-exp* [])
| (`mem` [])
| (`call` *fn* [])

Figure 7: Contexts for lifting embedded statements

statement in a non-pure expression and put it right after the program counter, thus making the original statement a little bit closer to being able to use one of the earlier rules. In the first case, if there is an `eseq`, the statement is lifted out and the `eseq` is replaced with just the expression portion. In the second case, when there is a `call`, the call is put into its own statement and the call is replaced by a register.

Figure 7 shows the context in which a flattening reduction can occur. The first case of flatten-S says that flattening can always occur in the first argument to a `move mem` expression. The second case says that a flattening reduction can occur inside the second argument to a `move mem` expression, but only if the first argument is a pure expression. This enforces a left-to-right evaluation order. That is the statements in the first argument will all have to be lifted out before the second case lets statements in the second argument be lifted out. Similarly for `cjump`. Otherwise, the grammar just allows statements to be lifted out anywhere an expression might occur.

The flatten-E1 context deserve special note. They define a single layer of a context where statments can be lifted out of expressions. Then, flatten-E is defined to either be a hole (i.e., a lifting can occur right at the top), or a single later context with another flatten-E inside it. Thus, flatten-E allows lifting arbitrarily deep in an expression. The flatten-E1 is needed in order to lift out `call` expressions. The [flatten-call] rule only lifts out a `call` when it is at least one layer deep (since if it is at the top already, then one of the earlier call rules should apply instead).