A Tiger Intermediate Language Specification

May 5, 2009

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-exps into pure-tree-exp. Figure[2] contains the definition of pure-tree-exp. They are just like tree-exps, 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 counter.
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 differ-
ence between it and the previous rule is that the eval-
uator 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 mem-
ory, 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 advanc-
ing the program counter).

The `[move-temp-alloc]` covers the case where the al-
location 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 mem-
ory address in the store and initializes the appropri-
ate 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.
\[ (S \text{tree-stm}_{\text{before}} ...) \rightarrow \text{MovePC}[[\text{Eval}][S, \text{pure-exp}]], \quad [\text{jump}] \]

\[ (S \text{tree-stm}_{\text{before}} ...) \rightarrow \text{MovePC}[[\text{Eval}][\text{pc}], \quad [\text{cjump-true}] \]

\[ (S \text{tree-stm}_{\text{before}} ...) \rightarrow \text{MovePC}[[\text{Eval}][\text{label}, \quad [\text{cjump-false}] \]

\[ \text{MovePC}[[\text{label}, (S \text{tree-stm}_{\text{before}} ... \text{label} \text{tree-stm}_{\text{after}} ...))] = (S \text{tree-stm}_{\text{before}} ... \text{label} \text{pc} \text{tree-stm}_{\text{after}} ...) \]

---

3 Jump rules

The jump rules are shown in figure 4. They hinge on the MovePC function. For \[ \text{jump} \], it evaluates the argument to \text{jump}, and then calls MovePC, supplying the value of \text{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 \text{cjump} rules evaluate the arguments to \text{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.

---

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
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 deserves special note. They define a single layer of a context where statements 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).