A Tiger Language Specification

May 5, 2009

```
exp ::= (biop exp exp)
      | (:= lvalue exp)
      | lvalue
      | num
      | str
      | nil
      | ()
      | (new id exp ...)
      | (new-array id exp exp)
      | (let (dec ...) exp)
      | (begin exp exp exp ...)
      | (when exp exp)
      | (while exp exp)
      | (if exp exp exp)
      | (for (id exp exp) exp)
      | (break)

dec ::= (var id exp)
      | (var id id exp)
      | (type id ty)
lvalue ::= id
      | (dot lvalue num)
      | (aref lvalue exp)
biop ::= relop | + | - | * | /
relop ::= eqop
      | <= | >= | < | >
eqop ::= = | <>
```

Figure 1: The Tiger language

$$(S\ E[(biop\ v_1\ v_2)]) \longrightarrow [\delta]\ (S\ E[\delta[[biop, v_1, v_2]]])$$

Figure 2: Delta rule

```
E ::= []
    | (biop E exp)
    | (biop v E)
    | (new id v ... E exp ...)
    | (new-array id E e)
    | (new-array id v E)
    | (let ([var id E] dec ...) exp)
    | (begin E exp exp ...)
    | (if E exp exp)
    | lval-E
    | (:= lval-E exp)
    | (:= lval-v E)
    | (loop E)
lval-v ::= id
    | (aref id num)
    | (dot id num)
lval-E ::= (aref E exp)
    | (aref v E)
    | (dot E exp)
    | (dot v E)
S ::= (fr ...)
fr ::= (id v)
    | (h:id (record v ...))
    | (h:id (array v ...))
v ::= num | str | nil | () | h:id
```

Figure 3: Evaluation contexts

1 Overview

This document describes an operational semantics for the Tiger language. The grammar of the language is shown in figure 1.

The semantics is described by a simplification rules that operate on a pair of an expression and a store. The store represents the memory of the machine and the expression represents the program. As the program is simplified, its evaluation may change the state of the machine. When it does, the store changes to reflect those changes.

The order in which an expression is evaluated is controlled by the evaluation contexts (figure 3). For example, arithmetic expressions are simplified leftmost, innermost first:

```
(* (+ 1 2) (+ 3 4)) → (* 3 (+ 3 4))
                    → (* 3 7)
                    → 21
```

Unlike exp, which specifies the shape of an expression in an outside-in fashion, E specifies the surroundings of some expression, in a kind of inside-out fashion. That is, an E describes an expression with a hole in the middle of it. The hole in the expression is writ-

ten []. So, the first clause tells us that the context might just be a hole. In other words, we've reached the middle point. The second clause says that the hole might be in the first subexpression of a biop expression. The second clause says that the hole might be in the second subexpression of a biop expression, but only if the first subexpression is a value (v). A value represents a fully simplified expression, and includes numbers, strings, `nil`, void (written `()`), and references into the store (ie, memory locations).

Consider specifically the first step in the reduction sequence above. It shows the simplification of `(+ 1 2)` to `3`. An equally good expression to simplify, however, would have been `(+ 3 4)` to `7`, but the reduction system will not allow that simplification, because the context of the reduction would have had to have been `(+ (+ 1 2) [])`, but that is not a valid E.

2 Delta

The first rule to consider is the δ rule, in figure 2 and it simply defers to a δ function that covers arithmetic and comparison operators. The δ function is intended to mimic the behavior of the machine's arithmetic operations on 32-bit signed numbers, i.e., modular arithmetic. The comparison operators return `1` if the comparison holds and `0` if it does not.

$(S\ E[(\texttt{begin}\ v\ exp)]) \longrightarrow (S\ E[exp])$ [begin2]

$(S\ E[(\texttt{begin}\ v\ exp_1\ exp_2\ exp_3\ ...)]) \longrightarrow (S\ E[(\texttt{begin}\ exp_1\ exp_2\ exp_3\ ...)])$ [beginN]

$(S\ E[(\texttt{if}\ 0\ exp_1\ exp_2)]) \longrightarrow (S\ E[exp_2])$ [if0]

$(S\ E[(\texttt{if}\ num\ exp_1\ exp_2)]) \longrightarrow (S\ E[exp_1])$ [ifN]

where non-zero?[[num]]

$(S\ E[(\texttt{when}\ exp_1\ exp_2)]) \longrightarrow (S\ E[(\texttt{if}\ exp_1\ exp_2\ ())])$ [when]

$(S\ E[(\texttt{for}\ (id\ exp_1\ exp_2)\ exp_3)]) \longrightarrow$ [for]

$(S\ E[(\texttt{let}\ ([\texttt{var}\ id\ exp_1]\ [\texttt{var}\ id_{top}\ exp_2])\ (\texttt{when}\ (< id\ id_{top})\ (\texttt{begin}\ exp_3\ (\texttt{while}\ (< id\ id_{top})\ (\texttt{begin}\ (:= id\ (+ id\ 1))\ exp_3)))))])$

where id_{top} fresh

$(S\ E[(\texttt{while}\ exp_1\ exp_2)]) \longrightarrow (S\ E[(\texttt{if}\ exp_1\ (\texttt{loop}\ (\texttt{begin}\ exp_2\ (\texttt{while}\ exp_1\ exp_2)))\ ())])$ [while]

$(S\ E_1[(\texttt{loop}\ E_2[(\texttt{break})])]) \longrightarrow (S\ E_1[()])$ [break]

where no-loop[[$E_2[(\texttt{break})]$]]

$(S\ E[(\texttt{loop}\ v)]) \longrightarrow (S\ E[v])$ [loop]

Figure 4: Control operator rules

3 Control

The rules in figure 4 govern the control operators in Tiger. The first two rules cover `begin` expressions. From the definition of E in figure 3, we know that the only place where evaluation occurs is in the first position of a `begin` expression so these rules kick in once that has been fully simplified. If the `begin` expression only has two subexpressions, the `begin` is just dropped by [begin0]. Otherwise, the first value is dropped by [beginN].

The `if` rules both drop the entire `if` expression, replacing it by the "then" or "else" portion, as appropriate, and `when` is treated as a shorthand for `if`.

The [for] rule replace a `for` expression by a while loop. This loop is a little more complex than you might expect in order to cope with the case where the value of upper bound of the loop is the maximum integer.

The last three rules cover `while` loops and `break`ing out of them. The basic idea is that `loop` is wrapped around the body of the loop as a marker to indicate where `break` should break to. Then, the [break] rule splits the context into two pieces and the side-condition ensures that there are no `loop` markers in the inner portion, so the entire inner portion is erased. Here is how an infinite `while` loop reduces[1]

```
   (while 1 ())
→ (if 1 (loop (begin () (while 1 ()))) ())
→ (loop (begin () (while 1 ())))
→ (loop (while 1 ()))
→ ...
```

[1]This is not an optimal reduction sequence because the terms grow forever. Challenge: develop an alternative rewriting strategy that avoids this infinite growth.

$((fr \ldots)\ E[(\texttt{let}\ ([\texttt{var}\ id_{old}\ v_1]\ dec \ldots)\ exp_2)]) \longrightarrow$ [letN]
$(((id_{new}\ v_1)\ fr \ldots)\ E[\mathsf{subst}[[id_{old}, id_{new}, (\texttt{let}\ (dec \ldots)\ exp_2)]]\ ])$
where id_{new} fresh

$(S\ E[(\texttt{let}\ ()\ exp)]) \longrightarrow$ [let0]
$(S\ E[exp])$

$(S\ E[(\texttt{let}\ ([\texttt{type}\ id\ ty]\ dec \ldots)\ exp)]) \longrightarrow$ [let-ty]
$(S\ E[(\texttt{let}\ (dec \ldots)\ exp)])$

$(S\ E[(\texttt{let}\ ([\texttt{var}\ id\ id_{ty}\ exp_r]\ dec \ldots)\ exp)]) \longrightarrow$ [let-idty]
$(S\ E[(\texttt{let}\ ([\texttt{var}\ id\ exp_r]\ dec \ldots)\ exp)])$

$((fr_{before} \ldots\ (id\ v)\ fr_{after} \ldots) \longrightarrow$ [get]
$E[id])$
$((fr_{before} \ldots\ (id\ v)\ fr_{after} \ldots)$
$E[v])$

$((fr_{before} \ldots\ (id\ v_{old})\ fr_{after} \ldots) \longrightarrow$ [set]
$E[(\texttt{:=}\ id\ v_{new})])$
$((fr_{before} \ldots\ (id\ v_{new})\ fr_{after} \ldots)$
$E[()])$

Figure 5: Let rules

With a break inside, however, we get this reduction sequence (where the ellipses elide a copy of the original loop):

```
   (while 1 (break))
→ (if 1 (loop (begin (break) ...)  ())
→ (loop (begin (break) ...))
→ ()
```

4 Let

The rules for `let` expressions manipulate the store. Each `var` binding in a let creates a new location in the store and saves its value there, as shown in the [letN] rule. The next three rules, [let0], [let-ty], and [let-idty] just clean up the other possible `let` expressions. The [get] rule looks up a value in the store when evaluation hits a variable reference. The [set] rule changes the value in the store when evaluation hits an assignment.

5 Arrays and records

The last group of rules cover how fields and arrays work. The rules for fields are very close to the rules

$((fr \ldots) \longrightarrow$ [new]
$E[(\texttt{new}\ id\ v \ldots)])$
$(((h{:}id\ (\texttt{record}\ v \ldots))\ fr \ldots)$
$E[h{:}id])$
where $h{:}id$ fresh

$((fr_{before} \ldots\ (h{:}id\ (\texttt{record}\ v_1 \ldots\ v\ v_2 \ldots))\ fr_{after} \ldots) \longrightarrow$ [dot]
$E[(\texttt{dot}\ h{:}id\ num)])$
$((fr_{before} \ldots\ (h{:}id\ (\texttt{record}\ v_1 \ldots\ v\ v_2 \ldots))\ fr_{after} \ldots)$
$E[v])$
where $num = \#(v_1 \ldots)$

$((fr_{before} \ldots\ (h{:}id\ (\texttt{record}\ v_1 \ldots\ v_{old}\ v_2 \ldots))\ fr_{after} \ldots) \longrightarrow$ [dot-set]
$E[(\texttt{:=}\ (\texttt{dot}\ h{:}id\ num)\ v_{new})])$
$((fr_{before} \ldots\ (h{:}id\ (\texttt{record}\ v_1 \ldots\ v_{new}\ v_2 \ldots))\ fr_{after} \ldots)$
$E[()])$
where $num = \#(v_1 \ldots)$

$((fr \ldots) \longrightarrow$ [new-array]
$E[(\texttt{new-array}\ id\ num\ v)])$
$(((h{:}id\ (\texttt{array}\ \mathsf{n\text{-}of}[[num, v]]\ )\ fr \ldots))$
$E[h{:}id])$
where $h{:}id$ fresh

$((fr_{before} \ldots\ (h{:}id\ (\texttt{array}\ v_1 \ldots\ v\ v_2 \ldots))\ fr_{after} \ldots) \longrightarrow$ [aref]
$E[(\texttt{aref}\ h{:}id\ num)])$
$((fr_{before} \ldots\ (h{:}id\ (\texttt{array}\ v_1 \ldots\ v\ v_2 \ldots))\ fr_{after} \ldots)$
$E[v])$
where $num = \#(v_1 \ldots)$

$((fr_{before} \ldots\ (h{:}id\ (\texttt{array}\ v_1 \ldots\ v_{old}\ v_2 \ldots))\ fr_{after} \ldots) \longrightarrow$ [aset]
$E[(\texttt{:=}\ (\texttt{aref}\ h{:}id\ num)\ v_{new})])$
$((fr_{before} \ldots\ (h{:}id\ (\texttt{array}\ v_1 \ldots\ v_{new}\ v_2 \ldots))\ fr_{after} \ldots)$
$E[()])$
where $num = \#(v_1 \ldots)$

Figure 6: Array and record rules

for arrays; the only difference being how the two constructs are allocated. A `new` expression reduces by creating a new location in the store and putting a record value there. The [dot] rule simplifies `dot` expressions by extracting the appropriate field from the record and the [dot-set] rule updates the store with a new binding for a field.

Like the [new] rule, the [new-array] rule creates a new location in the store, but this time binds it to an array and uses the n-of to fill in n copies of the initial value. The [aref] rule and the [aset] rules are identical to the [dot] and [dot-set] rules, except that they operate on records, not arrays.