Tiger-IR Reference Manual

January 17, 2019

1 Introduction

We will be using Tiger-IR, a quadruple three-address code-based intermediate representation in our projects. This document is a reference manual for the Tiger-IR language, a human readable language representation of Tiger-IR. For simplicity, we will not distinguish between Tiger-IR and the Tiger-IR language throughout the document.

A Tiger-IR program consists of one or more functions, one of which must be the main function. Each function contains a sequence of labels and instructions. Following is an example Tiger-IR program:

```
#start_function
int subtract_and_reset(int[2] x):
int-list: t0, t1, t2, s
float-list:
    array_load, t0, x, 0
    array_load, t1, x, 1
    sub, t2, t0, t1
    assign, s, t2
    array_store, 0, x, 0
    array_store, 0, x, 1
    return, s
#end_function

#start_function
void main():
int-list: p[2], t0, t1
float-list: f0
    callr, t0, geti
    callr, t1, geti
    callr, f0, getf
    array_store, t0, p, 0
    array_store, t1, p, 1
    callr, t0, subtract_and_reset, p
    call, puti, t0
    array_load, t1, p, 0
    call, puti, t1
    call, putc, 10
    brleq, if_label0, t0, 0
    call, putf, f0
    call, putc, 10
if_label0:
#end_function
```

The "`call, putc, 10`" instruction prints a `\n` character (`\n` is encoded as `10` in ASCII).

Following is an input/output pair of the above IR program:

```
Input:
3
2
1.0

Output:
1
0
1.0
```

2 Identifiers

Identifiers include the names of functions, variables, and labels. An identifier should be a sequence of one or more letters, digits, and underscores that does not start with a digit.

3 Data Types

There are two basic data types:

- `int`: 32-bit signed integer;
- `float`: single-precision (32-bit) floating point number.

There are also static array types whose element type can only be `int` or `float`. They are denoted as `int[N]` or `float[N]`, where `N` is the size of the array, which must be a constant positive integer.

4 Constants

There are two kinds of constants:

- Integer constant: a sequence starting with an optional "`-`", followed by one or more digits. Examples: `1`, `-2`.
- Float constant: a sequence starting with an optional "`-`", followed by one or more digits and a decimal point ("`.`"), with zero or more digits after the decimal point. Examples: `1.`, `-2.0`.

5 Functions

A function starts with a "`#start_function`" line and ends with an "`#end_function`" line.

In between, the first line is the function's signature in the following form:

```
ReturnType func_name(Type1 param1, Type2 param2, ..., Typen paramn):
```

`ReturnType` is the type of the function's return value. If the function does not return a value, it should be filled in with `void`.

The main function will have a function name of "main", with no parameter or return value.

The function signature is followed by a "data segment" consisting of the classification of all the variables (including temporaries, excluding parameters) inside the function based on their types. In particular, it consists of a list of variables belonging to the `int` type and another list of variables belonging to the `float` type. Arrays are classified to one of the lists by their element types, and they are denoted as `arr_name[N]`.

The format of the two lists is as follows:

```
int-list:  i1, i2, ..., im, ia1[N1], ia2[N2], ..., ian[Nn]
float-list:  f1, f2, ..., fp, fa1[N1], fa2[N2], ..., faq[Nq]
```

The order of variables in the lists does not matter. Arrays do not have to come after scalars.

Notes on semantics:
Function arguments with `int` or `float` type are passed by value. Array-typed arguments are passed by reference. The return value of a function cannot be array-typed. Two functions cannot have the same function name.

6 Labels and Instructions

In the body of a function, each non-empty line is a label or an instruction.

A label named "`label0`" will be displayed as the following text format:

```
label0:
```

We will be using quadruple three-address code in our project. The instructions (except function calls) will have the following form:

```
op, x, y, z
```

For binary arithmetic operators, this instruction can be viewed as "`x` ← `y op z`". For example, the source code expression, `2 * a + (b - 3)`, can be translated to the following IR instructions which computes the expression in `t3` (`t1`, `t2`, `t3` are temporary variables generated by the compiler when generating the IR):

```
mult, t2, a, 2
sub, t1, b, 3
add, t3, t1, t2
```

Some instructions may have less than three operands. Depending on the number of operands required, they will be represented as one of:

```
op, x, y
op, x
```

As a special case, function call instructions can have more than 3 operands. A function call with no return value will look like:

```
call, func_name, param1, param2, ..., paramn
```

And a function call with a return value received by a variable `x` will have the following form:

```
callr, x, func_name, param1, param2, ..., paramn
```

7 Instruction Reference

The tables below list all the instructions available in Tiger-IR, along with Tiger source code examples from which each instruction may be generated by a compiler front-end.

7.1 Assignment: `op, x, y`

Op	Example source	Example IR
assign	a := b	assign, a, b

The operands must be of the same basic type. The first operand must be a variable.

7.2 Binary Operation: `op, x, y, z`

Op	Example source	Example IR
add	a + b	add, t, a, b
sub	a - b	sub, t, a, b
mult	a * b	mult, t, a, b
div	a / b	div, t, a, b
and	a & b	and, t, a, b
or	a | b	or, t, a, b

The operands must be of the same basic type. The first operand must be a variable. Division by zero leads to undefined behaviors.

7.3 Goto: `op, label`

Op	Example source	Example IR
goto	break;	goto, after_loop

7.4 Branch: `op, label, y, z`

Op	Example source	Example IR
breq	if(a <> b) then	breq, after_if_part, a, b
brneq	if(a = b) then	brneq, after_if_part, a, b
brlt	if(a >= b) then	brlt, after_if_part, a, b
brgt	if(a <= b) then	brgt, after_if_part, a, b
brgeq	if(a < b) then	brgeq, after_if_part, a, b
brleq	if(a > b) then	brleq, after_if_part, a, b

The last two operands must be of the same basic type.

7.5 Return: `op, x`

Op	Example source	Example IR
return	return a;	return, a

The operand must be of a basic type. There is no empty return instruction. A function with no return value should not contain a return instruction.

7.6 Function call (no return value): `op, func_name, param1, param2, ..., paramn`

Op	Example source	Example IR
call	foo(x);	call, foo, x

The number, types and order of the arguments must match with the parameters of the target function.

7.7 Function call (with return value): `op, x, func_name, param1, param2, ..., paramn`

Op	Example source	Example IR
callr	a := foo(x, y, z);	callr, a, foo, x, y, z

The number, types and order of the arguments and the type of the first operand must match with the parameters and the return value of the target function. The first operand must be a variable but not an array.

7.8 Store into array: `op, x, array_name, offset`

Op	Example source	Example IR
array_store	arr[0] := a	array_store, a, arr, 0

The second operand must be an array. The type of the first operand must be the same as the element type of the second operand. The type of the third operand must be `int`.

7.9 Load from array: `op, x, array_name, offset`

Op	Example source	Example IR
array_load	a := arr[0]	array_load, a, arr, 0

The first operand must be a variable. The second operand must be an array. The type of the first operand must be the same as the element type of the second operand. The type of the third operand must be `int`.

7.10 Array Assignment: `op, x, size, value`

Op	Example source	Example IR
assign	type ArrayInt = array [100] of int; var X : ArrayInt := 10;	assign, X, 100, 10

Explanation: Assign 10 to all 100 elements in array `X`.

The first operand must be an array. The type of the second operand must be `int`. The type of the third operand must be the same as the element type of the first operand.

8 Intrinsic Functions

There are several intrinsic (builtin) functions that are handled specially by the compiler/interpreter. They are not declared/defined in the IR file, and their names are reserved in Tiger-IR. The list of intrinsic functions are given below:

Signature	Semantics
`int geti():`	`Read an integer from standard input.`
`float getf():`	`Read a float from standard input.`
`int getc():`	`Read a character as its ASCII value from standard input.`
`void puti(int i):`	`Print the integer i to standard output.`
`void putf(float f):`	`Print the float f to standard output.`
`void putc(int c):`	`Print the character encoded as the ASCII value c to standard output.`

`geti` and `getf` read an entire line of input up to and including the newline. Characters following the number are ignored. If the input line is invalid, `0` will be returned.