Tack Intermediate Representation Specification

Martin Hirzel and Robert Soulé

Version of November 3, 2015

Abstract

Tack is a simple statically-typed programming language invented for a Compiler Construction course. This document specifies the intermediate representation (IR) for Tack. The IR uses conventional 3-address instructions with an infinite supply of temporaries (i.e., virtual registers). It is similar to the IR in the Dragon book, see Section 6.2.1 of the 2nd edition, but with a few changes. Notably, the Tack IR includes source language types where needed for platform independence; adds instructions for casts and field accesses; and removes instructions for low-level address and pointer assignments. See also the Tack language specification: `http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/tack-spec.pdf`

1 Example: Hello, World!

```
main = fun () -> int
  param[0 : 1] = "Hello, world!\n";
  call print : 1;
  return 0;
```

Figure 1: A simple program in Tack IR.

Figure 1 shows an example program in Tack IR, consisting of a single function `main` with three instructions. Instruction `param` prepares an actual parameter for the subsequent function call, instruction `call` calls the intrinsic function `print`, and instruction `return` returns from `main`, exiting the program.

There is an interpreter available to test programs written in Tack IR. Use the following steps to try it:

- Download the interpreter from here: `http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/IRInterpreter.jar`
- Put the code from Figure 1 in a file `hello.ir`. The file extension `.ir` indicates a Tack IR file.
- Run the interpreter. The command-line is: `java -ea -jar IRInterpreter.jar hello.ir`

2 Tack IR Syntax

Figure 2 shows the grammar for Tack IR. A program consists of one or more function definitions. Each function definition has a name, a function type, and one or more statements. Each statement consists of zero or more labels and an instruction. The function type is specified using the *funType* non-terminal, which is defined in the Tack language specification.

3 Addresses

We refer to the operands of instructions as "addresses". This is common practice in compilers and computer architecture, even though it is slightly misleading, because addresses are not necessarily pointers. Tack IR uses the following kinds of addresses:

- *nameAddr*, such as `x`, is the identifier of a local variable or function parameter.

// Top-level grammar:

program → *funDef*$^+$

funDef → **id** = `fun` *funType stmt*$^+$

stmt → *labelColon** *instr* `;`

labelColon → *label* `:`

label → **id**

// Addresses, i.e., operands for instructions:

addr → *nameAddr* | *constAddr* | *tempAddr* | *sizeofAddr*

nameAddr → **id**

constAddr → **boolLit** | **intLit** | **stringLit** | `null`

tempAddr → **id**

sizeofAddr → `sizeof (` *type* `)`

// Instructions:

instr → *copyInstr* | *infixInstr* | *prefixInstr* | *castInstr*
| *uncondJumpInstr* | *trueJumpInstr* | *falseJumpInstr* | *relopJumpInstr*
| *paramInstr* | *callInstr* | *returnInstr*
| *arrReadInstr* | *arrWriteInstr* | *recReadInstr* | *recWriteInstr*

// Instructions for expressions computing values:

copyInstr → *addr* = *addr*

infixInstr → *addr* = *addr* **infixOp** *addr*

infixOp → `+` | `-` | `*` | `/` | `%`

prefixInstr → *addr* = **prefixOp** *addr*

prefixOp → `-`

castInstr → *addr* = *addr* `:` *type*

// Instructions for statements and boolean jumping code:

uncondJumpInstr → `goto` *label*

trueJumpInstr → `if` *addr* `goto` *label*

falseJumpInstr → `ifFalse` *addr* `goto` *label*

relopJumpInstr → `if` *addr* **relOp** *addr* `goto` *label*

relOp → `==` | `!=` | `>=` | `>` | `<=` | `<`

// Instructions for function calls and returns:

paramInstr → `param [` *index* `:` *arity* `]` = *addr*

index → **intLit**

arity → **intLit**

callInstr → `call` **id** `:` *arity*
| *addr* = `call` **id** `:` *arity*

returnInstr → `return`
| `return` *addr*

// Instructions for accessing memory:

arrReadInstr → *addr* = *addr* `[` *addr* `]`

arrWriteInstr → *addr* `[` *addr* `]` = *addr*

recReadInstr → *addr* = *addr* `.` **id**

recWriteInstr → *addr* `.` **id** = *addr*

Figure 2: Tack IR grammar.

- *constAddr*, such as `"Hello, world!\n"`, is a primitive TACK literal, using the corresponding token syntax from the TACK language specification.
- *tempAddr*, such as `t3`, is the identifier of a compiler-generated temporary variable. For example, intermediate results of arithmetic expressions such as `(x+1)/2` are kept in temporaries.
- *sizeofAddr*, such as `sizeof((i:int,j:int))`, is the size needed for a value of the specified TACK type. This is typically used as a parameter to one of the intrinsics `newRecord` or `newArray`. Using *sizeofAddr* makes the IR more platform-independent than providing concrete integers for sizes.

4 Instructions

There are 15 IR instructions, consisting of the 4 groups shown in Figure 2:

- *Computing values.* These instructions only include arithmetic operators, no boolean operators. The assumption is that arithmetic expressions use these instructions here, whereas boolean expressions use jumping instructions.
- *Statements and jumping code.* Each of these instructions uses a label as a jump target. The target label must be attached to an instruction in the same function as the jump instruction.
- *Function calls and returns.* Each *callInstr* is preceded by one *paramInstr* for each parameter. The *arity* is the total number of parameters. Figure 1 shows a simple example, calling the `print` intrinsic function with a single parameter. For a list of all intrinsics, see Section 7 of the TACK language specification.
- *Accessing memory.* These instructions read and write data in arrays and records. Each instruction only accesses one "level" of a data structure. Multi-level data accesses require multiple instructions.

5 Conclusions

The TACK compiler front-end translates TACK source code to TACK IR. And the TACK compiler back-end translates TACK IR to assembly code. The class project has milestones for both the front-end and the back-end. In our class, we implement a back-end to x64 assembly, although it would be equally possible to go to other target languages. If you want a preview for what the back-end needs to do, see the x64 introduction on the class webpage: `http://www.inf.usi.ch/faculty/soule/teaching/2015-fall/cc/x64-intro.pdf`