Prof. Dr. Peter Thiemann
Matthias Keil

Winter Semester 2016/2017

Compiler Construction

`http://proglang.informatik.uni-freiburg.de/teaching/compilerbau/2016ws/`

Exercise Sheet 5

1 From Spiglet to Kanga (15 + 10 Points)

The simplified AST in Spiglet serves as a good base for data flow analyses. Next step: register allocation! To this end, we define another intermediate language, Kanga, where temporaries are mapped to registers or spilled to the stack frame. Kanga is already fairly close to MIPS assembler, our final target language. Its grammar and semantics is defined as follows.

Grammar

Program	::=	`MAIN [` *IntegerLiteral* `] [` *IntegerLiteral* `] [` *IntegerLiteral* `]` *StmtList* `END` *Procedure**
StmtList	::=	(Label$^?$ *Stmt*)*
Procedure	::=	*Label* `[` *IntegerLiteral* `] [` *IntegerLiteral* `] [` *IntegerLiteral* `]` *StmtList* `END`
Stmt	::=	`NOOP`
	|	`ERROR`
	|	`CJUMP` *Reg Label*
	|	`JUMP` *Label*
	|	`HSTORE` *Reg IntegerLiteral Reg*
	|	`HLOAD` *Reg Reg IntegerLiteral*
	|	`MOVE` *Reg Exp*
	|	`PRINT` *SimpleExp*
	|	`ALOAD` *Reg SpilledArg*
	|	`ASTORE` *SpilledArg Reg*
	|	`PASSARG` *IntegerLiteral Reg*
	|	`CALL` *SimpleExp*
Exp	::=	`HALLOCATE` *SimpleExp*
	|	*Operator Reg SimpleExp*
	|	*SimpleExp*
Operator	::=	`LT`
	|	`PLUS`
	|	`MINUS`
	|	`TIMES`
SimpleExp	::=	*Reg*
	|	*IntegerLiteral*
	|	*Label*

SpilledArg	::=	`SPILLEDARG` *IntegerLiteral*
Reg	::=	`a0` | `a1` | `a2` | `a3`
	|	`t0` | `t1` | `t2` | `t3` | `t4` | `t5` | `t6` | `t7` | `t8` | `t9`
	|	`s0` | `s1` | `s2` | `s3` | `s4` | `s5` | `s6` | `s7`
	|	`t8` | `t9`
	|	`v0` | `v1`
IntegerLiteral	::=	⟨integer literal⟩
Label	::=	⟨identifier⟩

Semantics

The semantics of Kanga is in most cases equivalent to the constructs in Spiglet. However, the following changes hold:

Labels

All labels have global scope, and should therefore be unique.

Registers

Instead of an unlimited number of temporaries with local scope, Kanga has 24 global machine registers with global scope. The registers `s0-s7` and `t0-t9` can be allocated for general use. As with MIPS, the `s` registers are callee-saved, whereas all other registers are not preserved across calls. Registers `a0-a3` are reserved to pass arguments to a procedure call. Register `v0` is reserved to return a result from a procedure call, and `v0` and `v1` are also used as temporary registers when values need to be loaded from the stack.

Stack

Values can be loaded from and stored to the stack with the `ALOAD` and `ASTORE` instructions. Here, `SPILLEDARG` *i* denotes the ith value on the stack, the first value can be found at `SPILLEDARG` 0.
Example:

```
ALOAD  s3 SPILLEDARG 1
```

loads the second value from the stack into register `s3`.

Procedures

A procedure has now three integers in its header, e.g. procA [5] [3] [4].

The first integer denotes (as in Spiglet) the number of arguments taken by the procedure.

The second integer is for the number of stack slots that the procedure requires. This is the total number of all stack slots needed, including space for arguments (if necessary, cf. Calls), space for any spilled temporaries, and space for any registers that have to be saved.

The third integer is the maximum number of arguments of any call in the body of the procedure. For example, if procA makes a call to procB that takes 3 arguments, to procC that takes 2 arguments, and to procD that takes 4 arguments, then since 4 is the maximum number of arguments a call in the body of procA uses, this integer is set to 4.

Further, a procedure body is no longer a *StmtExp* but a *StmtList*. The return value is expected to be put in register `v0`.

Calls

Call is now a statement. As mentioned above, registers `a0-a3` are used to send arguments. If the called procedure takes more than 4 arguments, you need to use the `PASSARG` stmt, which saves the extra arguments to the stack. For whatever reason, `PASSARG` starts at position 1, but `SPILLEDARG` starts at 0, so in general an argument passed as `PASSARG` i is accessed in the body of the procedure as `SPILLEDARG` $i-1$.

Example: Consider a call to some procedure with label P and arguments stored in registers `t1`, `t2`, `t3`, `t4`, and `t5`. The return value should go in `t6`.

```
MOVE a0 t1      // First move 4 args to the ”a” registers.
MOVE a1 t2
MOVE a2 t3
MOVE a3 t4
PASSARG 1 t5    // If there are more args, save them to the stack.
                // Note that PASSARG is 1-based, not 0-based!
CALL P
MOVE t6 v0      // The return value will be in v0.
```

The project template contains a Kanga interpreter which tries hard to discover problems as early as possible: it prevents procedures from reading the caller's registers (except for a0–a3), it detects procedures which return without having restored callee-save registers (except for MAIN), and it clears *all* caller-save registers (except for v0,v1) on return. This improves error detection in simple register allocators at the cost of making interprocedural register allocation impossible.

Project - Part 5

Implement an AST transformation from Spiglet to Kanga. This exercise consists of two major parts:

Liveness Analysis (5 + 10 Points)

- Specify a (intra-procedural) liveness analysis for Spiglet.
  - In Spiglet, each procedure basically consists of a list of possibly labeled statements. Each statement is uniquely identified by its position in the list. For a procedure $S_0 \dots S_n$, we define

    $$pos(S_0 \, \dots \, S_n) = \{0, \dots, n\}$$

    to be the set of all possible positions within a procedure. We define as the entry point of the procedure

    $$initial(S_0 \, \dots \, S_n) = 0$$

    and further as the exit point

    $$final(S_0 \, \dots \, S_n) = n$$

    The flow equations are now a mapping of labeled statement (with position) to a set of two positions denoting an edge in the flow graph.
    Example:

    $$flow([l : \texttt{NOOP}]^i) = \{(i, i+1)\} \qquad \forall i \in \{0, \dots, n-1\}$$

    Specify in a similar way the flow equations for all statement types in Spiglet.
  - For each statement, specify the use and def sets of temporaries. You may want to define auxiliary definitions for expressions etc.

  The formal specification should be submitted as a PDF. An example for such a specification can be found in *Nielson et al., Principles of Program Analysis, Springer 2005*, Chapter 2.1 and 2.1.4. (There, use sets are called gen sets, and defs sets are called kill sets. Further, the labels they use do not correspond to labels in Piglet, but to the position of a statement in the statement list of a procedure.)

- Implement the construction of the flow graph and the liveness analysis for Spiglet. Test your liveness analysis thoroughly; debugging bad liveness information by observing misbehaving Kanga programs is *not* fun.

Register allocation (10 Points)

- Implement some register allocation algorithm for Spiglet and a transformation of Spiglet code to Kanga.
- Remember that each method call requires some prologue and epilogue for caller and callee. For Kanga, you need to take care of arguments, caller-save and callee-save registers, and the return value. (The interpreter handles the stack pointer).

- To simplify things, you may assume that the interference graph of the temporaries is always colorable, i.e. spilling and coalescing is not needed. (Beware, you will still need to spill arguments when a procedure call requires more than 4 arguments!)
- Write a (short) description of your register allocation.
- On the homepage, you will find a test environment with Kanga parser, Kanga pretty-printer and Kanga interpreter.
- As always, be careful when adding nodes to SableCC ASTs: don't add one node twice to a tree.

Bonus Task: Spilling and coalescing (up to 10 Points)

- Change your register allocation algorithm such that it can also spill if necessary.
- Improve it further by coalescing source and destination nodes of move instructions.
- To get points for this task, you are required to have already implemented the simplified register allocation correctly!

Submission

- Deadline: **02.02.2017, 12:00 (noon)**. Late submissions will not be accepted.
- Submit your solution to the subversion repository. Your submission will consist of one folder (exercise5) which includes your solution.
- Rewrite method `spiglet.tokanga.SpigletToKangaTranslator.translateProgram` so that it calls your Kanga transformation for the given Spiglet AST.
- Your solution will consist of: 1. a zip file as generated by `ant submission` with the implementation, and 2. a pdf `registeralloc-<your name>.pdf` with the specification of the analyses and a description of the implemented register allocation.
- You are strongly encouraged to test your solution with the provided test data. Add test cases as you might think necessary. You need not submit your own test cases.
- The description must be limited to ten pages. Submitting more than ten pages will lead to reduction in points.
- The description may be either German or English. Clear and understandable style is required.