Compiler Construction

Integers and Variables
Chapter 2

Hannes Saffrich

University of Freiburg
Department of Computer Science
Programming Languages

23. April 2024

The Plan

- Big Picture
  - Start with a compiler and interpreter for $\mathcal{L}_{var}$ (Chapter 2)
  - Extend the compiler and interpreter with new features in the rest of this course
- Today (lecture)
  - Writing an interpreter (without parser)
  - Introduction to RISC-V Assembly
  - Dicussion of the $\mathcal{L}_{var}$ compiler passes (Chapter 2)
- Friday (lecture)
  - Lexing and Parsing (Chapter 3)
- Next Tuesday (tutorial)
  - Introduction to the exercise framework
  - Introduction to the first exercise:
    - Writing a compiler for the $\mathcal{L}_{var}$ language

Interpreter

- Let's write an interpreter for $\mathcal{L}_{var}$!

$$\xrightarrow{\text{Text}} \text{Lexer} \xrightarrow{\text{Tokens}} \text{Parser} \xrightarrow{\text{Py AST}} \text{Type Checker} \xrightarrow{\text{Py AST}} \text{Evaluator}$$

The $\mathcal{L}_{var}$ language (Chapter 2)

$$
\begin{aligned}
\langle prog \rangle &::= \langle stmt \rangle^* \\
\langle stmt \rangle &::= \langle expr \rangle \\
&\mid \langle var \rangle \ \texttt{=} \ \langle expr \rangle \\
&\mid \texttt{print(}\langle expr \rangle\texttt{)} \\
\langle expr \rangle &::= \langle int \rangle \\
&\mid \langle var \rangle \\
&\mid \langle op_1 \rangle \langle expr \rangle \\
&\mid \langle expr \rangle \langle op_2 \rangle \langle expr \rangle \\
&\mid \texttt{input\_int()} \\
\langle op_1 \rangle &::= \texttt{-} \\
\langle op_2 \rangle &::= \texttt{-} \mid \texttt{+}
\end{aligned}
$$

Example Program:

```
x = input_int()
y = input_int() + x
print(x + y - 5)
3 + 4
input_int()
```

RISC-V Example

- Example: The C program

```
int main(void) {
    return 42;        // Exit with exit code 42
}
```

  corresponds to this RISC-V assembly code:

```
    .globl main
main:
    li a0, 42
    ret
```

- Generally, an assembly program is a list of
  - assembly instructions, which are translated to one or more machine code instructions
  - labels, which give a name to the address of the next instruction, and can be used in control flow instructions like call main or j main.
  - assembly directives, which specify metadata for later compiler passes like machine code generation and linking

RISC-V Registers

- Registers are the internal memory of a processor
- RISC-V 64 provides 32 registers to store integers
- Each register stores 64 bit of data
- Some registers have special meaning, e.g.
  - the `zero` register is *hardwired* to always contain the constant 0
  - the `sp` register is used *by convention* to store the stack pointer
- In RISC architectures, all instructions operate on registers, except for special *load* and *store* instructions which transfer data between RAM and registers.
- In CISC architectures, also other instructions can directly refer to addresses in RAM, e.g. addition.

RISC-V Registers

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

The RISC-V Instruction Set Manual, Chapter 20, p. 109
`https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`

RISC-V Basic Instructions

- Accessing RAM
  - The *load* instruction transfers data from RAM to a register, e.g.

    ```
    ld a0, -16(fp)
    ```

    loads the data from RAM address `fp - 16` to register `a0`.
  - The *store* instruction transfers data from a register to RAM, e.g.

    ```
    sd a0, -16(fp)
    ```

    stores the data from register `a0` at RAM address `fp - 16`.
- Storing a constant in a register
  - The *load immediate* instruction loads a constant into a register, e.g.

    ```
    li a0, 42
    ```

RISC-V Basic Instructions

- Integer Arithmetic
  - The *add* instruction adds the data from two registers and stores the result in a third register, e.g.

    ```
    add a0, a1, a2
    ```

    adds `a1` and `a2` and stores the result in `a0`.
  - For subtraction, multiplication, division and modulo there are similar instructions called `sub`, `mul`, `div`, and `rem`, respectively.
  - The *addi* instruction adds the data from a register to a constant and stores the result in another register, e.g.

    ```
    addi a0, a1, 42
    ```

    adds 42 to `a1` and stores the result in `a0`. As the constant is part of the instruction, it has to be within the bounds of a 12-bit integer.
  - There are *no* immediate instructions for subtraction, multiplication, division and modulo.

RISC-V Basic Instructions

- Function Calls
  - The `call label` instruction calls a function by
    - writing the address of the next instruction (`pc + 4`) into the return address register `ra`
    - setting the program counter `pc` to the address described by `label`
  - The `ret` instruction returns from a function by
    - setting the program counter to the address stored in the return address register `ra`
  - For `ret` to return to the right place, the return address register `ra` needs to contain the same value from when the function was called
  - What if our function calls other functions?
    - A convention dictates if the caller or callee is responsible for saving a register.
    - This is described in "Saver" column of the register table
    - For the return address register `ra`, it is the responsibility of the caller to save the register
    - This means if we call a function, then that function is allowed to change the content of `ra`, so if we still need the content of `ra` after the call, we need to save it before the call in a callee-save register or on the stack.

RISC-V Basic Instructions

- Function Calls
  - Arguments and return values are *not* part of the `call` and `ret` instructions
  - Instead, they are stored in registers or on the stack
  - A *calling convention* dictates where exactly they have to be placed:
    - 64-bit integer arguments are stored in the registers `a0-a7`.
    - Return values are stored in registers `a0` and `a1`.
    - If more than 8 arguments are passed, they are stored on the end of the caller's stack frame in descending order, i.e. argument 9 at `0(sp)`, argument 10 at `8(sp)`, etc.
    - For larger arguments (e.g. C-structs) different rules can apply (not important for us now)
    - The stack pointer register `sp` points to the beginning of the last word (8 bytes) of the caller's stackframe.
    - The frame pointer points to the beginning of the last word *before* the current stackframe. Usually, this is the stack pointer of the caller.
    - If return address and/or frame pointer have to be saved, then they are saved at the beginning of the stack frame, and return address comes before frame pointer.

RISC-V Basic Instructions

- Example
  - Function `foo` calls function `bar` with 11 integer arguments
  - Function `bar` uses three local variables, which are stored on the stack
  - The following shows the stack at the time when execution is inside `bar`:

Frame	Position	Position	Contents
`bar`	`0(sp)`	`-48(fp)`	Empty for alignment
	`8(sp)`	`-40(fp)`	Local Var 3
	`16(sp)`	`-32(fp)`	Local Var 2
	`24(sp)`	`-24(fp)`	Local Var 1
	`32(sp)`	`-16(fp)`	`foo`'s `fp`
	`40(sp)`	`-8(fp)`	Return Address
`foo`	`48(sp)`	`0(fp)`	Argument 9
	`56(sp)`	`8(fp)`	Argument 10
	`64(sp)`	`16(fp)`	Argument 11

RISC-V Basic Instructions

- Example
  - The assembly code creating this stack frame could look as follows:

```
foo:
    ...
    li a0, ARG1
    li a1, ARG2
    ...
    li a7, ARG8
    li t0, ARG9
    sd t0, 16(sp)
    li t0, ARG10
    sd t0, 8(sp)
    li t0, ARG11
    sd t0, 0(sp)
    call bar
    ...
```

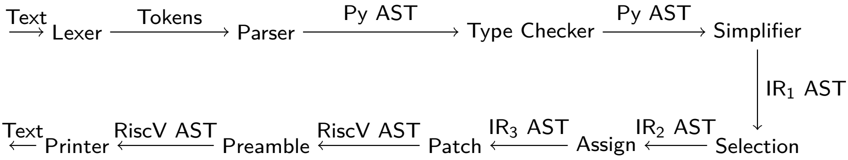
```
bar:
    sd ra, -8(sp)
    sd fp, -16(sp)
    addi fp, sp, 0
    addi sp, sp, -48
    li t0, LOCAL1
    sd t0, -24(fp)
    li t0, LOCAL2
    sd t0, -32(fp)
    li t0, LOCAL3
    sd t0, -40(fp)
    ... # <- you are here
    li a0, RESULT
    addi sp, sp, 48
    ld ra, -8(sp)
    ld fp, -16(sp)
    ret
```

RISC-V Basic Instructions

- Function Calls
  - It is possible to deviate from the calling convention, if you generate the code for both caller and callee.
  - It is important to follow the calling convention, when calling C-functions, which is necessary to use operating system functionality, e.g. file I/O, printing to the terminal, or network access.

$\mathcal{L}_{var}$ Compiler

- Goal: compile $\mathcal{L}_{var}$ programs to RISC-V 64 assembly
- Multiple passes and intermediate languages
- C Runtime for `input_int` and `print` functions
- Use `gcc` to generate machine code from assembly and link with the machine code of the runtime



$\mathcal{L}_{var}$ Compiler: Monadic Normalform

- Instructions don't have subexpressions
- The $\mathcal{L}_{var}$ language does have arbitrarily nested subexpressions
- Idea: Assign subexpressions to new temporary variables
- Example:

```
x = input()
print((x + 3) -
    5)
```

$\Longrightarrow$

```
x = input()
tmp:0 = x + 3
tmp:1 = tmp:0 - 5
print(tmp:1)
```

- Output is a program in the IR language $\mathcal{L}_{var}^{mon}$, which is like $\mathcal{L}_{var}$, but expressions must have variables or constants as subexpressions.

$\mathcal{L}_{var}$ Compiler: Instruction Selection

- Transform $\mathcal{L}_{var}^{mon}$ programs to $riscv_{var}$ programs.
- Example:

```
x = input()

tmp:0 = x + 3
tmp:1 = tmp:0 - 5
print(tmp:1)
```

$\Longrightarrow$

```
call input_int64
mv #x, a0
add #tmp:0, #x, 3
sub #tmp:1, #tmp:0, 5
mv a0 #tmp:0
call print_int64
```

$\mathcal{L}_{var}$ Compiler: Assign Homes

- Transform $riscv_{var}$ programs to $riscv_{mem}$ programs.
- Example:

```
call input_int64
mv #x, a0
add #tmp:0, #x, 3
sub #tmp:1, #tmp:0, 5
mv a0 #tmp:0
call print_int64
```

$\Longrightarrow$

```
call input_int64
mv -24(fp), a0
add -32(fp), -24(fp), 3
sub -40(fp), -32(fp), 5
mv a0, -40(fp)
call print_int64
```

$\mathcal{L}_{var}$ Compiler: Patch Instructions

- Transform $riscv_{mem}$ programs into actual RISC-V 64 programs.
- Example:

`call input_int64`		`call input_int64`
`mv -24(fp), a0`		`add t0,zero,a0`
		`sd t0,-24(fp)`
`add -32(fp), -24(fp), 3`		`ld t1,-24(fp)`
		`addi t0,t1,3`
	$\Longrightarrow$	`sd t0,-32(fp)`
`sub -40(fp), -32(fp), 5`		`ld t1,-32(fp)`
		`addi t0,t1,-5`
		`sd t0,-40(fp)`
`mv a0, -40(fp)`		`ld a0,-40(fp)`
`call print_int64`		`call print_int64`

$\mathcal{L}_{var}$ Compiler: Add Prelude and Conclusion

- Transform the RISC-V 64 program into a RISC-V 64 program.
- Example:

```
    .globl main
main:
    sd ra,-8(sp)
    sd fp,-16(sp)
    addi fp,sp,0
    addi sp,sp,-48
    call input_int64
    add t0,zero,a0
    sd t0,-24(fp)
    ld t1,-24(fp)
    addi t0,t1,3
    sd t0,-32(fp)
    ld t1,-32(fp)
    addi t0,t1,-5
    sd t0,-40(fp)
    ld a0,-40(fp)
    call print_int64
    addi a0,zero,0
    addi sp,sp,48
    ld ra,-8(sp)
    ld fp,-16(sp)
    ret
```

$\mathcal{L}_{var}$ Compiler: Runtime

- Implemented in C

```
#include <stdio.h>

void print_int64(long x) {
    printf("%ld\n", x);
}

long input_int64() {
    long x = 0;
    scanf("%ld", &x);
    return x;
}
```

- On `RISC-V 64` a `long` is a 64-bit integer.

$\mathcal{L}_{var}$ Compiler: Runtime

- Cross-platform alternative:

```
#include <stdint.h>
#include <inttypes.h>
#include <stdio.h>

void print_int64(int64_t x) {
    printf("%" PRId64 "\n", x);
}

int64_t input_int64() {
    int64_t x = 0;
    scanf("%" SCNd64, &x);
    return x;
}
```

$\mathcal{L}_{var}$ Compiler: Running our assembly

- Use *gcc* variant for cross-compilation to RISC-V 64
- Compile our assembly and link together with our runtime:
  `riscv64-linux-gnu-gcc-10 -static foo.S runtime.c -o foo`
- Use *qemu* to emulate the RISC-V program on your local machine:
  `qemu-riscv64-static foo`
- We provide a `Dockerfile` containing both the RISC-V *gcc* and *qemu*