SOM2IF15 – Compilation

Activation Records and Intermediate Representation

Frédéric Loulergue

2022

Outline

Activation Records / Frames

Frames during Execution

- Activation record = frame
- Stack of frames to store local variables, intermediate results, etc. for *block*
- Other names: call stack, execution stack, program stack
- In high-level presentations: for both inline blocks and function bodies

Frames during Compilation

- data structure containing information on functions
- used to generate code to manage the call stack during execution

Frames

Inline blocks / Function bodies

- Having activation records for both inline blocks and function bodies is conceptually correct and simpler than having activation records for function bodies only
- In the implementation however, frames are only created for functions
- The stack structure is needed only because of *recursion*: it is not possible to re-enter in an inline block without first exiting the block if there isn't any recursive call.
- Recursive call = need for a new *function* activation record

Structure of Functions

Version 1

```
int abs(int x){
  if (x < 0)
    return -x
  else
    return x
}
```

Version 2

```
int abs(int x){
  var int result
  if (x < 0)
    result = -x
  else
    result = x
  return result
}
```

Function in the Intermediate Representation

- one entry point
- one exit point

Registers

Intermediate Representation

- No variable declaration
- An infinite member of temporary registers

Functions

- parameters replaced by temporary registers
- if there is a returned result: in other temporary register

Frames

- during code generation, if a parameter is passed by reference, it is not possible to use a register (we need the address, a register doesn't have an address)
- during code generation, we need the size of each (execution) frame

The Frame Data Structure in the LUO Compiler

Frame

entryLabel	label of the entry point of the function's body
returnLabel	label of the exit point
parameters	list of temporary registers storing the parameters
result	a temporary register storing the result, if needed
size	size necessary to store the execution frame

Outline

Intermediate Representation

Expressions

$\langle expr \rangle$	$::=$	*number*	literal number
	$\mid$	$\texttt{reg}_i$	temporary register
	$\mid$	$\texttt{reg}_i[\langle expr \rangle]$	memory read
	$\mid$	*unop* $\langle expr \rangle$	unary operation function application
	$\mid$	$\langle expr \rangle$ *binop* $\langle expr \rangle$	binary operation application

where

- $i \in \mathbb{Z}$
- *unop* contains `-`, `!`, `length`, all conversion functions
- *binop* contains all arithmetic and Boolean binary operations but NOT `++` and `--`

Intermediate Representation

Instructions

$\langle com \rangle$	::=	*label*:	a label
	|	$\texttt{reg}_i := \langle expr \rangle$	write to a register
	|	$\texttt{reg}_i + \langle expr \rangle := \langle expr \rangle$	write in memory
	|	`jump` $(\langle expr \rangle)$ *label*, *label*	conditional jump
	|	`goto` *label*	unconditional jump
	|	`call` *frame* $\langle expr \rangle, \ldots, \langle expr \rangle$	function/system call
	|	$\texttt{reg}_i :=$ *frame* $\langle expr \rangle, \ldots, \langle expr \rangle$	function/system call

Remarks

Frames in calls

- In LUO, calls can be *statically* resolved
- In a language like Java, it is not possible: to know which method to call, we need the class of the object, known only at *runtime*

Temporary Registers and Labels

- We assume we can generate always generate a fresh temporary register and a fresh label

Types

- We assume the LUO code has been type-checked
- Do we need type information here?

Remarks

Frames in calls

- In LUO, calls can be *statically* resolved
- In a language like Java, it is not possible: to know which method to call, we need the class of the object, known only at *runtime*

Temporary Registers and Labels

- We assume we can generate always generate a fresh temporary register and a fresh label

Types

- We assume the LUO code has been type-checked
- Do we need type information here?
- Yes, at least for the memory space: `byte`, `int`, `address`

Outline

Intermediate Representation

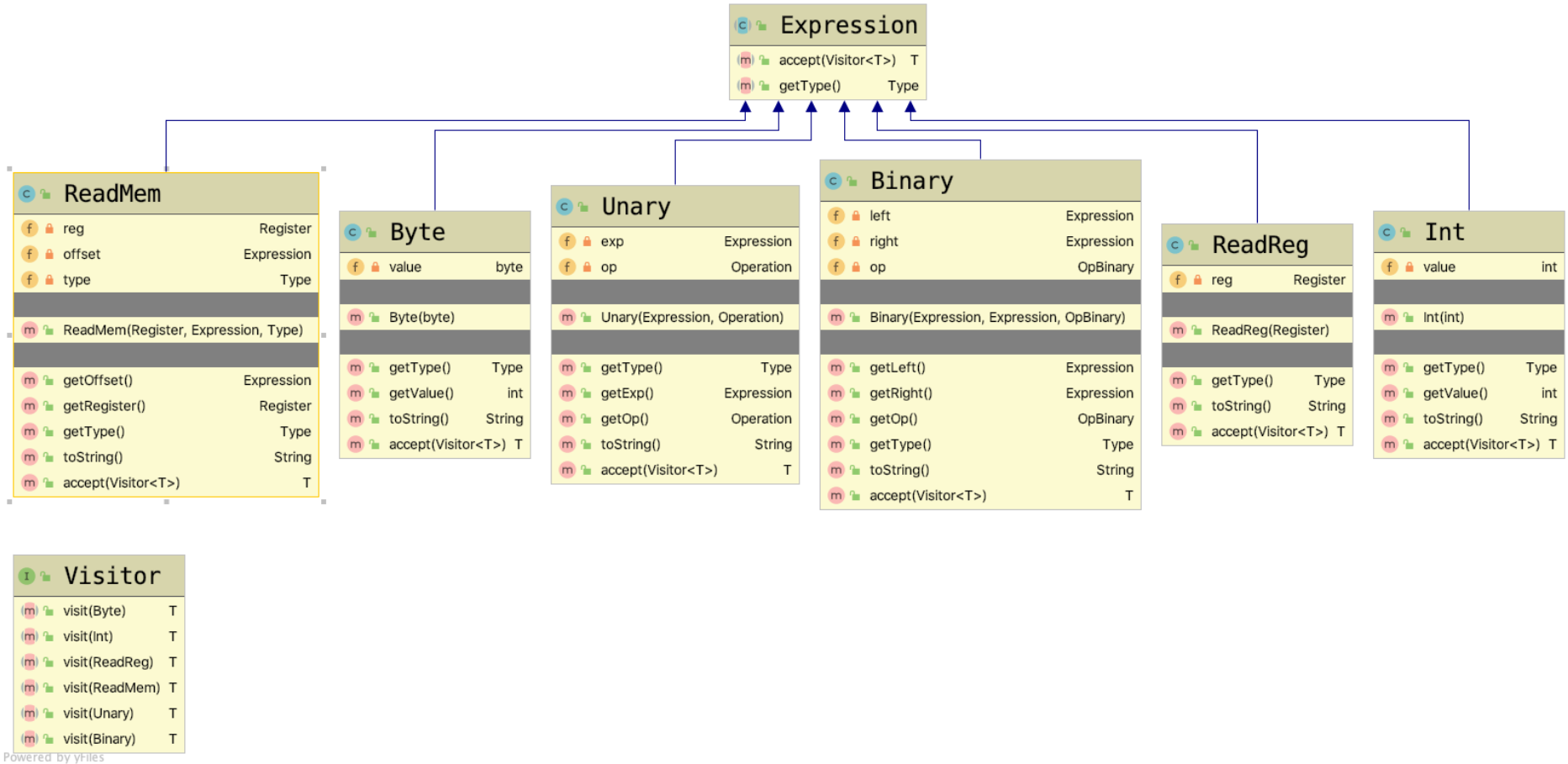
Expressions

$$
\begin{array}{rcll}
\langle expr \rangle & ::= & number & \text{literal number} \\
& | & \texttt{reg}_i & \text{temporary register} \\
& | & \texttt{reg}_i[\langle expr \rangle] & \text{memory read} \\
& | & unop\ \langle expr \rangle & \text{unary operation function application} \\
& | & \langle expr \rangle\ binop\ \langle expr \rangle & \text{binary operation application}
\end{array}
$$

where

- $i \in \mathbb{Z}$
- *unop* contains `-`, `!`
- *binop* contains all arithmetic and Boolean binary operations but NOT `++` and `--`

Intermediate Representation in Java

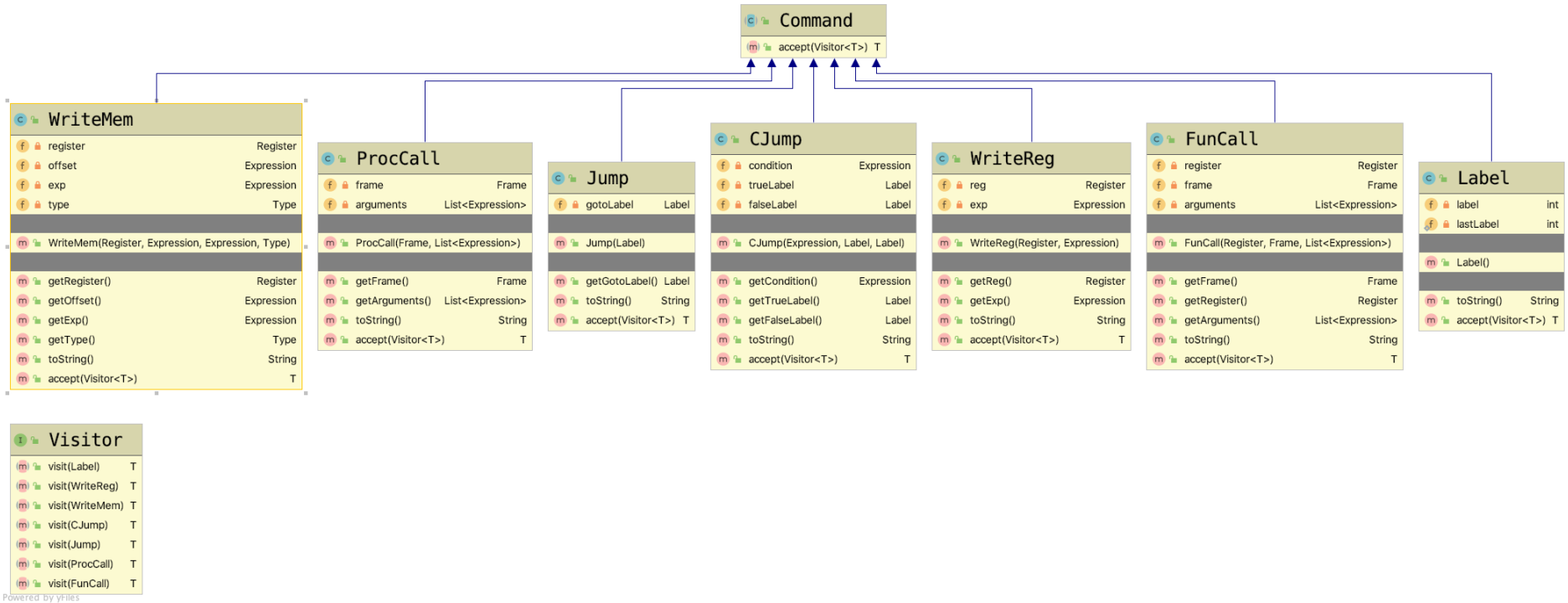


Intermediate Representation

Instructions

$\langle com \rangle$	::=	*label*:	a label
	|	$\texttt{reg}_i := \langle expr \rangle$	write to a register
	|	$\texttt{reg}_i + \langle expr \rangle := \langle expr \rangle$	write in memory
	|	`jump` $(\langle expr \rangle)$ *label*, *label*	conditional jump
	|	`goto` *label*	unconditional jump
	|	`call` *frame* $\langle expr \rangle, \ldots, \langle expr \rangle$	function/system call
	|	$\texttt{reg}_i :=$ *frame* $\langle expr \rangle, \ldots, \langle expr \rangle$	function/system call

Intermediate Representation in Java



Outline

Translation of LUO Expressions

Variable

- Each time a variable is declared (`Declaration`) it is associated with a *fresh* temporary register (in a `Map<String, Register>`)
- When translating an `ExpVariable`, we just lookup the register in the map

Recursive Construction

For a `ExpBinaryOperation`:

- we compile the left argument,
- we compile the right argument,
- and we build an `ir.expr.Binary` using the pieces.

Not so simple:

- some LUO expressions are not IR expressions!
- they should be translated as a mix of IR expressions and commands

Translation of LUO Expressions

LUO expressions without an IR correspondence

- `ExpFunctionCall` they are IR *commands*
- `ExpArrayAccess` / `ExpRecordAccess`: `MemWrite` is close but the "array"/"record" part should be a register, while it is an arbitrary expression in LUO
- `ExpNew`, `ExpString`, `ExpArrEnum`: see TP
- `ExpAssignop`: can be removed by an AST transformation, unsupported

Translation of LUO Expressions

Function Calls (User or Predefined)

Principle of the transformation in LUO Syntax:

- Initial: `add(x, 1) + 2`
- Translated: `int tmp = add(x,1)`
  and the new expression is tmp + 2

In IR, assuming $\texttt{x} \mapsto \texttt{reg}_0$ and $\texttt{add} \mapsto \textit{Frame}_0$:

- LUO: `add(x, 1) + 2`
- IR command part: $\texttt{reg}_1 := \texttt{call}\ \textit{Frame}_0\ \texttt{reg}_0,\ 1$
- IR expression part: $\texttt{reg}_1 + 2$

Translation of LUO Expressions

How do we know that `add` $\mapsto$ $Frame_0$?

- User-defined function: Frames are built by a first visitor (inner class `FramesBuilder`)
- Pre-defined functions + `new` + `print` + `read`: TP

Translation of LUO Expressions

Array Access

- Same principle
- A new register is introduced: contains an address
- A new `WriteReg` is introduced
- LUO: `m[0][1]`
- Assuming $\texttt{m} \mapsto \texttt{reg}_0$:
  - Command part: $\texttt{reg}_1 := \texttt{reg}_0[0]$
  - Expression part: $\texttt{reg}_1[1]$

Translation to Intermediate Representation in Java

```
public class Result
{
    private Expression expression;
    private List<Command> code;

    public Expression getExp() { return expression; }
    public List<Command> getCode() { return code; }

    public Result(Expression expression, List<Command> code){
        this.expression = expression;
        this.code = code;
    }
    public Result(Expression expression) {
        this(expression, new LinkedList<>());
    }
    public Result(List<Command> code) {
        this(null, code);
    }
}
```

Translation of LUO Statements

Declaration

- Variable associated with a *fresh* register
- It there is an initialization:
  - compilation of the expression
  - write of this expression to the new register

Translation of LUO Statements

Return Statement

- Compilation of the expression
- Jump to the exit label of the frame
- It requires we know what's the current frame: field `currentFrame`

Translation of LUO Statements

Assignment Statement

- In LUO: `x = 1` and `a[0] = 1` and record
- In IR: two different commands: `WriteReg` and `WriteMem`
- For `WriteMem`: the address should be in the form:

$$\texttt{reg} + \langle \textit{expr} \rangle$$

Translation of LUO Statements

Other Statements

Next TP

Conditional

$$\texttt{if (expr)}\ block_1\ \texttt{else}\ block_2 \implies [\ \texttt{jump}(\mathcal{C}(b))\ L_1,\ L_2;\ L_1 :;\ \mathcal{C}(block_1);\ \texttt{goto}\ L_3\ L_2 :;\ \mathcal{C}(block_2);\ \texttt{goto}\ L_3\ L_3 :\ ]$$

where L_1, L_2 and L_3 are fresh labels.

While Loops

- Same principles as conditional