



Compiler Construction

Lecture 19: Code Generation V (Compiler Backend)

Summer Semester 2016

Thomas Noll

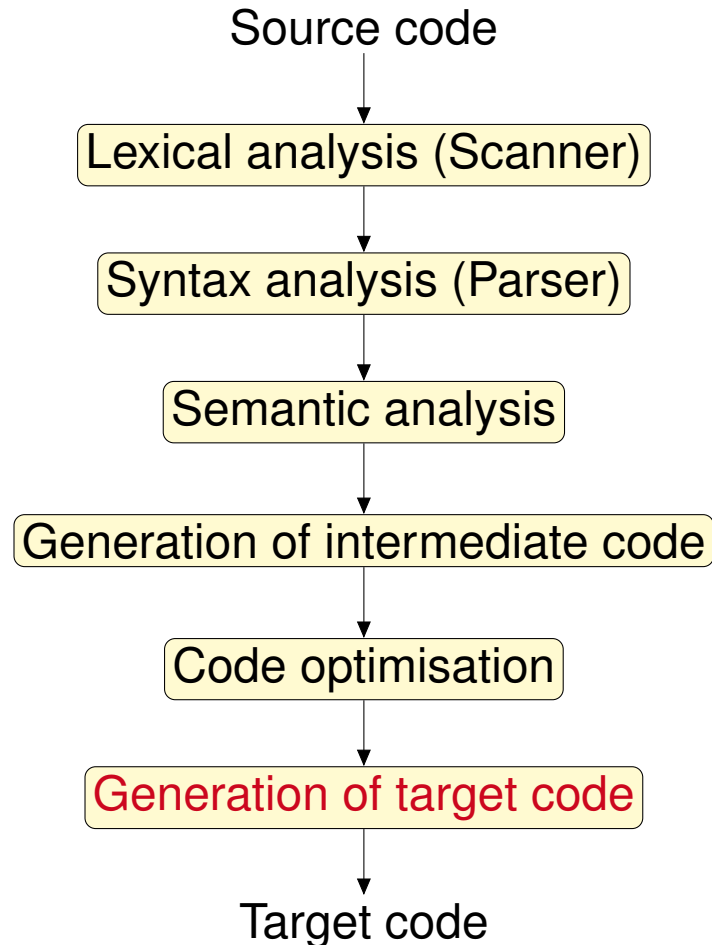
Software Modeling and Verification Group

RWTH Aachen University

<https://moves.rwth-aachen.de/teaching/ss-16/cc/>

The Compiler Backend

Conceptual Structure of a Compiler



The Compiler Backend

The Compiler Backend

Final step: **translation** of (optimised) abstract machine code into “real” machine code
(possibly followed by assembling phase)

Goal: **runtime and storage efficiency**

- fast backend
- fast and compact code
- low memory requirements for data

Memory hierarchy: **decreasing speed & costs**

- registers (program counter, data [universal/floating point/ address], frame pointer, index register, condition code, ...)
- cache (“fast” RAM)
- main memory (“slow” RAM)
- background storage (disks, sticks, ...)

Principle: use **fast memory** whenever possible

- evaluation of expressions in registers (instead of data/runtime stack)
- code/procedure stack/heap in main memory

Instructions: select adequately (number/type of operands, addressing modes, ...)

Code Generation Phases

1. **Register allocation:** registers used for
 - values of (frequently used) variables and intermediate results
 - computing memory addresses (array indexing, ...)
 - passing parameters to procedures/functions
2. **Instruction selection:**
 - translation of abstract instructions into (sequences of) real instructions
 - employ special instructions for efficiency (e.g., `INC(x)` rather than `ADD(x, 1)`)
3. **Instruction scheduling (placement):** increase level of parallelism and/or pipelining by smart ordering of instructions

Register Allocation

Register Allocation

Example 19.1

Assignment:

$$z := (u+v) - (w - (x+y))$$

Target machine with

r registers R_0, R_1, \dots, R_{r-1}
and main memory M

Instruction types:

$$R_j := M[a]$$
$$M[a] := R_j$$
$$R_j := R_j \text{ op } M[a]$$
$$R_j := R_j \text{ op } R_j$$

(with address a)

Instruction sequence ($r = 2$):

$$R_0 := M[u]$$
$$R_0 := R_0 + M[v]$$
$$R_1 := M[x]$$
$$R_1 := R_1 + M[y]$$
$$M[t] := R_1$$
$$R_1 := M[w]$$
$$R_1 := R_1 - M[t]$$
$$R_0 := R_0 - R_1$$
$$M[z] := R_0$$

Shorter sequence:

$$R_0 := M[w]$$
$$R_1 := M[x]$$
$$R_1 := R_1 + M[y]$$
$$R_0 := R_0 - R_1$$
$$R_1 := M[u]$$
$$R_1 := R_1 + M[v]$$
$$R_1 := R_1 - R_0$$
$$M[z] := R_1$$

- **Reason:** 2nd variant avoids **intermediate storage** t for $x+y$
- How to compute **systematically**?
- **Idea:** start with **register-intensive** subexpressions

Register Allocation

Register Optimisation

- Let $e = e_1 \text{ op } e_2$.
- Assumption: e_i requires r_i registers for evaluation
- Evaluation of e :
 - if $r_1 < r_2 \leq r$, then e can be evaluated using r_2 registers:
 1. evaluate e_2 (using r_2 registers)
 2. keep result in 1 register
 3. evaluate e_1 (using $r_1 + 1 \leq r_2$ registers in total)
 4. combine results
 - if $r_2 < r_1 \leq r$, then e can be evaluated using r_1 registers
 - if $r_1 = r_2 < r$, then e can be evaluated using $r_1 + 1$ registers
 - if more than r registers required: use main memory as intermediate storage
- The corresponding optimisation algorithm works in two phases:
 1. Marking phase (computes r_i values)
 2. Generation phase (produces actual code)(cf. Wilhelm/Maurer: *Übersetzerbau, 2. Auflage*, Springer, 1997, Sct. 12.4)

Register Allocation

The Marking Phase

Algorithm 19.2 (Marking phase)

Input: *expression*
(with binary operators *op* and variables *x*)

Procedure: *recursively compute*

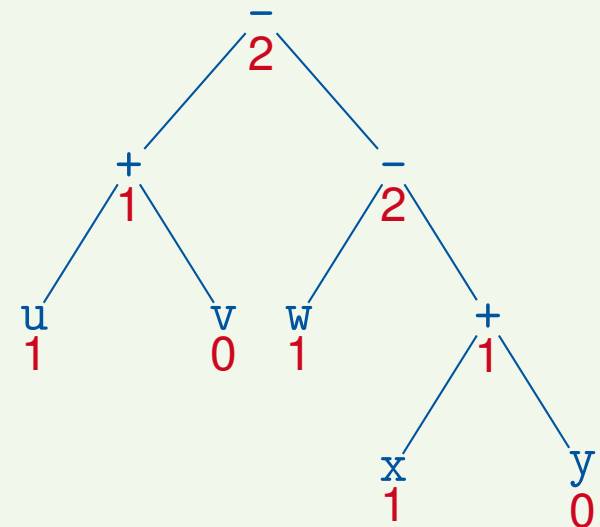
$$r(x) := \begin{cases} 1 & \text{if } x \text{ is a "left leaf"} \\ 0 & \text{if } x \text{ is a "right leaf"} \\ 1 & \text{if } x \text{ is at the root} \end{cases}$$

$$r(e_1 \text{ op } e_2) := \begin{cases} \max\{r(e_1), r(e_2)\} & \text{if } r(e_1) \neq r(e_2) \\ r(e_1) + 1 & \text{if } r(e_1) = r(e_2) \end{cases}$$

Output: *number of required registers* $r(e)$

Example 19.3 (cf. Ex. 19.1)

$e = (u+v) - (w - (x+y))$:



Register Allocation

The Generation Phase I

- **Goal:** generate optimal (= shortest) code for evaluating expression e with register requirement $r(e)$
- **Data structures** used in Algorithm 19.4:
 - RS : stack of available registers
(initially: all registers; never empty)
 - CS : stack of available main memory cells
- **Auxiliary procedures** used in Algorithm 19.4:
 - $output$: outputs the argument as code
 - top : returns the topmost entry of a stack S (leaving S unchanged)
 - pop : removes and returns the topmost entry of a stack
 - $push$: puts an element onto a stack
 - $exchange$: exchanges the two topmost elements of a stack

Register Allocation

The Generation Phase II

Algorithm 19.4 (Generation phase)

Input: expression e , annotated with register requirement $r(e)$

Variables: RS : stack of registers; CS : stack of memory cells; R : register; C : memory cell;

Procedure: recursive execution of procedure $code(e)$, defined by $code(e) :=$

- (1) if $e = x$, $r(x) = 1$: % left leaf
 $output(top(RS) := M[x])$
- (2) if $e = e_1 op y$, $r(y) = 0$: % right leaf
 $code(e_1)$;
 $output(top(RS) := top(RS) op M[y])$
- (3) if $e = e_1 op e_2$, $r(e_1) < r(e_2)$, $r(e_1) < r$:
 $exchange(RS)$;
 $code(e_2)$;
 $R := pop(RS)$;
 $code(e_1)$;
 $output(top(RS) := top(RS) op R)$;
 $push(RS, R)$;
 $exchange(RS)$
- (4) if $e = e_1 op e_2$, $r(e_1) \geq r(e_2)$,
 $r(e_2) < r$:
 $code(e_1)$;
 $R := pop(RS)$;
 $code(e_2)$;
 $output(R := R op top(RS))$;
 $push(RS, R)$
- (5) if $e = e_1 op e_2$, $r(e_1) \geq r$, $r(e_2) \geq r$:
 $code(e_2)$;
 $C := pop(CS)$;
 $output(M[C] := top(RS))$;
 $code(e_1)$;
 $output(top(RS) := top(RS) op M[C])$;
 $push(CS, C)$

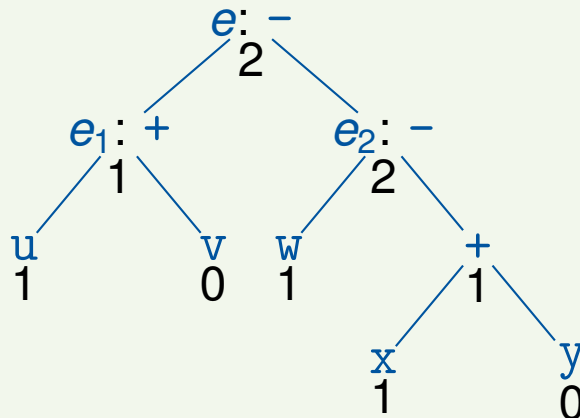
Output: optimal (= shortest) code for evaluating e

Register Allocation

The Generation Phase III

- **Invariants** of Algorithm 19.4:
 - after executing $code(e)$, both RS and CS have their original values
 - after executing $code(e)$, value of e is stored in topmost register of RS
- **Shortcoming** of Algorithm 19.4: multiple evaluation of **common subexpressions** (\implies dynamic programming [Wilhelm/Maurer])

Example 19.5 (cf. Example 19.3)



Application of Algorithm 19.4:
on the board

Register Allocation

Register Allocation by Graph Coloring

- Algorithm 19.4: register allocation for single expressions
- Required: global allocation within program/procedure body
- Approach: **graph coloring**

Register Allocation by Graph Coloring

1. Use unbounded number of **symbolic registers** for storing intermediate values
2. Consider life span of symbolic registers: r is **live** at program point p if
 - there is a path to p on which r is set and
 - there is a path from p on which r is read before being set
3. **Life span** of r = program points where r is live
4. Two registers are in **collision** if one is set in the life span of the other
5. Yields **register collision graph** (nodes = life spans, edges = collisions)
6. Program executable with k real registers iff collision graph k -colorable

Further Topics in Compiler Construction

- Translation of **higher-level constructs** (modules, classes, ...)
- Translation of **non-procedural languages**
 - object-oriented (polymorphism, dynamic dispatch)
 - functional (higher-order functions, type checking/inference)
 - logic (unification, backtracking)
- Code **optimisation**
- **Symbol-table handling**
- **Error handling**
- **Bootstrapping**

Exams & Seminars

Exams

1. Thursday, 28 July, 10:00–13:00, AH 1/2/6, 5056
2. Tuesday, 6 September, 10:00-13:00, 10:00-13:00, AH 2/3

Winter Semester 2016/17: *Analysis and Verification of Pointer Programs*

- Automata-based: regular model checking, forest automata
- Logic-based: shape analysis, separation logic
- Graph-based: graph grammars, graph transformation systems
- Extensions for concurrency

Companion seminar: *Probabilistic Programs* [J.-P. Katoen, M. Volk et al.]

Outlook

Lectures

Winter Semester 2016/17: *Static Program Analysis*

- Dataflow analysis
- Abstract interpretation
- Interprocedural analysis
- Pointer analysis

Summer Semester 2017: *Semantics and Verification of Software*

- Operational semantics
- Denotational semantics
- Axiomatic semantics
- Semantic equivalence
- Compiler correctness