Optimizing Compilers

Bernhard Scholz
Institut f. Computersprachen
Argentinierstr. 8
scholz@complang.tuwien.ac.at

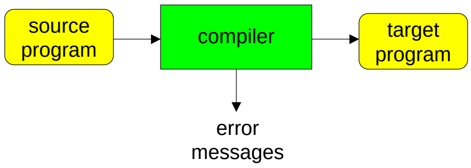
Objectives

- Overview of optimizing compilers
- Program analysis and transformations
- Algorithms and data structures for performing analysis and transformation
- *Hands-on exercise:*
  write your own optimizing compiler, i.e. ***MiniC***

References

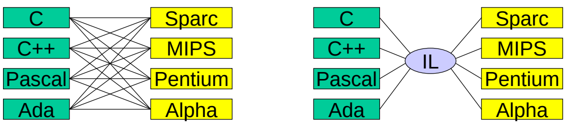
- Material for this course
  `www.complang.tuwien.ac.at/scholz/lecture/`
- Material for Übersetzerbau
  `www.complang.tuwien.ac.at/ublu/`
- Books
  Appel: Modern compiler implementation
  Aho, Sethi, Ullman: Compilers
  Muchnik: Advanced Compiler Design & Implementation
  Zima: Supercompilers for Parallel and Vector Computing

Compilers



- Optimizing compiler
  - translation considering specific objectives, e.g. run-time, code-size, power-consumption
  - conflicting goals

Intermediate Language(IL)



- IL is an abstract machine language on high level
- Decouple front end from back end
  - without IL
    - n languages, m targets $\Rightarrow n \times m$ compilers
  - with IL
    - n front ends, m back ends
- Problem: loss of high-level information

Intermediate Language(2)

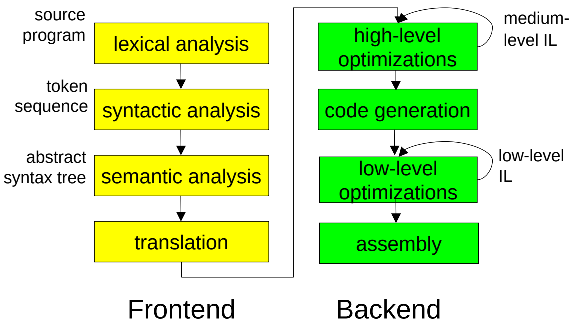
- High-level
  - quite close to source language
  - e.g. abstract syntax tree.
  - code generation issues are quite clumsy in high-level IL
- Medium-level
  - have some low-level features for code-generation
  - represent source variables, temporaries, and registers
  - reduce control flow to conditional and unconditional branches.
  - adequate IL to perform machine-independent optimizations
- Low-level
  - correspond to target-machine instruction

Our Intermediate Language(OIL)

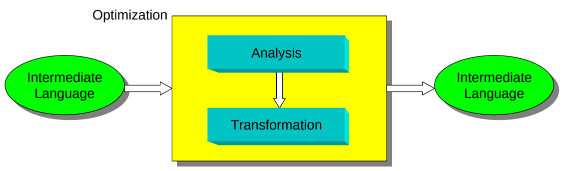
- only one data type (word)
- no global variables
- functions
  - parameters
  - return value
  - no declaration for local variables
- Statements
  - assignments
  - branches
  - function calls
  - return statements

```
// factorial number
fac(x)
{
 b = x <= 1;
 if (b) goto L1;
 a = x - 1;
 r = fac(a);
 r = x * r;
 return r;
L1:
 return 1;
}
```


Optimizing Compilers



Optimization



- Analysis
  - properties of programs
  - safe, pessimistic assumptions (input & paths not known a priori)
- Transformation: based on Analysis

A Brief Optimization Taxonomy

- Context
  - expression (statement level/local)
  - basic block (local)
  - procedure (intra-procedural)
  - whole program (inter-procedural)
- Type
  - static (without runtime information)
  - feed-back (with runtime information)
  - dynamic (during runtime)

Optimization Examples

- algebraic simplification: x+0
- constant propagation: x=2; ...; y = 2+x;
- common sub-expressions: x=(a*b)/c; y=(a*b)*2;
- dead variables: x=(a+b); …; x = 5;
- copy propagation: x = y; …; z = x;
- dead code: b=0; if(b) ....
- code motion: if(b) x=(a+b); else x=(a+b);
- function inlining: int inc(i) {return i+1;}

Basic Blocks

- unit of translation, i.e. important data structure
- sequence of consecutive statements
- enters at the beginning and leaves at the end

Algorithm:

1. Determine the set of leaders
   - First statement of a function
   - Any statement that is the target of a conditional or unconditional jump
   - Any statement that immediately follows a goto, conditional jump, or return statement
2. For each leader: all statements up to but not including the next leader or the end of the function.

Example for Basic Blocks

Intermediate code

```
// factorial number
fac(x)
{
 b = x <= 1;
 if (b) goto L1;
 a = x - 1;
 r = fac(a);
 r = x * r;
 return r;
L1:
 return 1;
}
```

Basic blocks of code

B1:
```
 b = x <= 1;
 if (b) goto L1;
```

B2:
```
 a = x - 1;
 r = fac(a);
 r = x * r;
 return r;
```

B3:
```
L1:
 return 1;
```


Function Calls in Basic Blocks

- Can call instructions cause a problem?
  - in most cases: need not be considered
- Fortran:
  - alternate return can be programmed
  - therefore: a call might be a basic block boundary
- C:
  - features inter-procedural control-flow
  - setjump/longjump
  - watch out, this might have nasty side-effects
- Pascal:
  - goto-statements leaving procedure-boundaries
  - simplified setjump/longjump version

Control-Flow Graph (CFG)

- Fundamental data structure
  - for inter-procedural optimizations,
  - for data flow analysis
- Control-flow graph
  - rooted directed graph with nodes and edges
  - nodes are basic blocks
  - edge represents flow of control
  - two unique nodes: start/end node
  - add artificial end node if several exits exist

Example for Control-Flow Graph

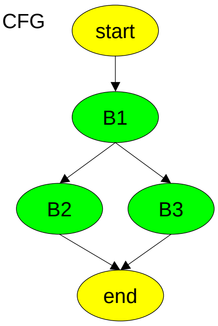
Basic blocks

B1:
```
b = x <= 1;
if (b) goto L1;
```

B2:
```
a = x - 1;
r = fac(a);
r = x * r;
return r;
```

B3:
```
L1:
 return 1;
```

CFG

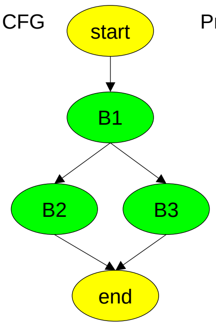


Control-Flow Graph(2)

- CFG is a directed graph $G\langle N,E,start,end\rangle$
  - N set of nodes (basic blocks)
  - $E \in N \times N$ set of edges
  - start node *start*
  - end node *end*
- Predecessors
  $preds(x) = \{u \mid (u,x) \in E\}$
- Successors
  $succs(x) = \{u \mid (x,u) \in E\}$
- Start /end node properties
  $preds(start) = \{\}$ (start node has no predecessors)
  $succs(end) = \{\}$ (end node has no successors)

Example(2)

CFG



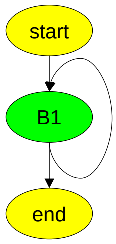
Predecessors and Successors

u	*preds*(*u*)	*succs*(*u*)
start	{}	{*B1*}
B1	{*start*}	{*B2, B3*}
B2	{*B1*}	{*end*}
B3	{*B1*}	{*end*}
end	{*B2, B3*}	{}

Paths in Control-Flow Graphs

- Path
  - sequence $\pi \in N^k$ of nodes
  - nodes are connected by an edge
- Example
  - $\langle start, B1, end \rangle$, $\langle start, B1, B1, end \rangle$
  - $\langle start, end \rangle$ is not a path
- Definition
  - A sequence π of nodes $\langle u_1, u_2, \ldots, u_k \rangle$ is a *path* iff $(u_i, u_{i+1}) \in E$ for all $0<i<k$.
  - A *program path* is path $\langle start, u_2, \ldots, end \rangle$.

CFG



Unreachable-Code Elimination

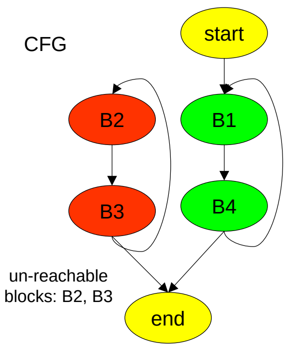
- Problem
  - not all basic blocks are reachable from start node
  - cannot be possibly be executed
  - there are no paths from the entry to the block
- Effect
  - no direct effect on the execution speed
  - decreases code-size
- Analysis
  - compute for all blocks reachability, i.e. exists a path from start node to a node
  - reverse analysis result for obtaining unreachable blocks
- Transformation:
  - remove all unreachable basic blocks

Example for UC-Elimination

Basic blocks of code

B1:
```
L0:
  b = x <= 1;
  goto L3;
```
B2:
```
L1:
  b = x <= 1;
  goto L2;
```
B3:
```
L2:
  a = x - 1;
  r = x * r;
  if(b) goto L1;
```
B4:
```
L3:
  if(b) goto L0;
```

CFG



Analysis for UC

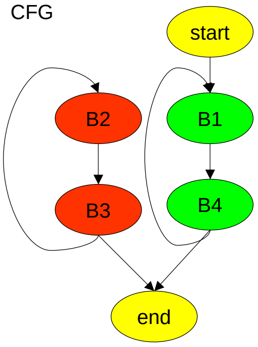
- Approach
  - work-list **W**
  - reachable set of nodes **R**
  - node *n* is reachable if $n \in \mathbf{R}$
  - basic block delete-able if $n \notin \mathbf{R}$
- Complexity
  O(|N|^2)

Algorithm

```
R = ∅
W = {start}
repeat
  R = R ∪ W;
  for all n ∈ W do
   W=(W ∪ succs(n))-R
  end for
until W = ∅;
```


Example for UC-Elimination(2)

CFG



- Iteration steps
  1. W={start} R={}
  2. W={B1} R={start}
  3. W={B4} R={start,B1}
  4. W={end} R={start,B1,B4,end}
  5. W={} R={start,B1,B4,end}
- Remark
  B1 is not added to W in step 4
  (is already in R!)
- Unreachable blocks
  N-R={B2,B3}

Stop

- Next lecture: 20.3.2003, 13:45 – 14:45
- Assignment: due to 20.3.2003