Advanced compiler construction

Michel Schinz
2006-03-17

General course information

Teachers

Teacher:

Michel Schinz

Michel.Schinz@epfl.ch

Assistant:

Iulian Dragos

INR 321, ☎ 368 68 64

Iulian.Dragos@epfl.ch

Course goal

The goal of this course is to teach you:

1. how to compile high-level (functional and OO) programming languages, and
2. how to optimise the generated code.

Outline

The course is split in two main parts, of approximately the same length.

Part I: compilation of high-level languages (virtual machines, memory management, closure conversion, etc.)

Part II: optimisations (data-flow analysis, SSA form, etc.)

Evaluation

Evaluation will be based on:

- a project, made in groups of two persons,
- an individual oral exam at the end of the semester.

Notice that the oral exam will take place **in the last week** of the semester, not after it.

Project

You will have to improve a compiler and a virtual machine (VM) for `minischeme` a tiny dialect of Scheme, itself a dialect of Lisp. Example:

```
(define map
  (lambda (f l)
    (if (null? l)
        nil
        (cons (f (head l))
              (map f (tail l))))))
```

The compiler is written in Scala, the VM in C.

Project parts

The project will contain two parts:

1. a common part, where all groups will have to complete the same "simple" tasks (*e.g.* add automatic memory management to the VM),
2. an individual part, where all groups will have to choose one advanced task, try to complete it and describe their work in a short report (*e.g.* implement a JIT compiler for the VM).

Web resources

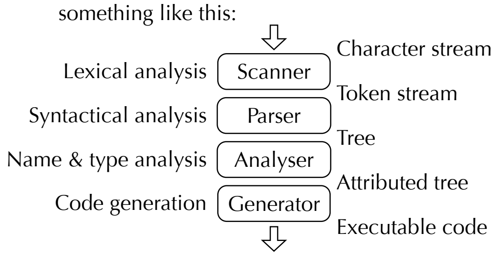
The course has a Web page:
`http://lamp.epfl.ch/teaching/advancedCompiler/2006/`

Moreover, we will use Moodle to handle the project:
`http://moodle.epfl.ch/`

Course overview

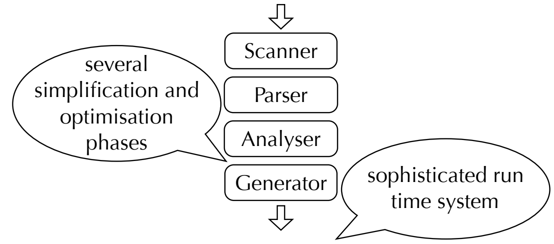
What is a compiler?

Your current view of a compiler must be something like this:



What is a compiler?

Real compilers are often more complicated…



Simplification and optimisation phases

Simplification phases transform the program so that complex concepts (*e.g.* pattern matching, closures, …) are translated using simpler ones.

Optimisation phases transform the program so that it – hopefully – makes better use of some resource (*e.g.* CPU cycles, memory, etc.).

Of course, all these phases must preserve the semantics of the original program!

Intermediate representations

Simplification and optimisation phases must represent the program in some way.

One possibility is to use the representation of the parser – the abstract syntax tree (AST).

The AST is perfectly suited to certain tasks, but other **intermediate representations** (IR) exist and are more appropriate in some situations.

Kinds of intermediate representations

Intermediate representations can broadly be split in three categories:

- **graphical IRs** which represent the program as a graph,
- **linear IRs** which represent the program as a linear sequence of instructions, and
- **hybrid IRs**, which are partly graphical, partly linear.

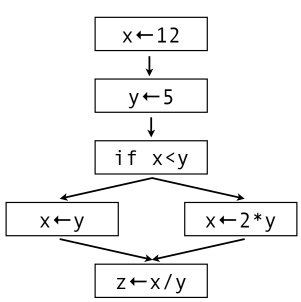
Graphical IRs

Graphical IRs represent the program as a graph.

They are often used in the initial phases of the compiler. In particular, the AST produced by the parser is a graphical IR.

Examples: ASTs, some kinds of control-flow graphs (CFG), etc.

Graphical IR example



Linear IRs

Linear IRs represent the program as a sequence of instructions.

They are often used in the final phases of the compiler, since machine code itself is linear.

Examples: three-address code, stack languages.

Linear IR example

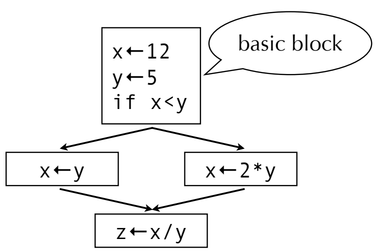
```
    x←12
    y←5
    if x<y goto L1
    x←y
    goto L2
L1: x←2*y
L2: z←x/y
```

Hybrid IRs

Hybrid IRs have graphical and linear components.

For example, most control-flow graphs are hybrid: the nodes in the CFG are linear sequences of instructions – called basic blocks – but the CFG itself is a graph.

Hybrid IR example



SSA form

Static single-assignment (**SSA**) form is an IR with an important characteristic: all "variables" are assigned exactly once.

This characteristic makes a lot of optimisations easier. For example, identifying common sub-expressions is trivial.

Transforming an imperative program to SSA form implies the introduction of so-called **Φ-functions**.

SSA example

```
    x1←12
    y1←5
    if x1<y1 goto L1
    x2←y1
    goto L2
L1: x3←2*y1
L2: x4←Φ(x2,x3)
    z1←x4/y1
```

The Φ-function "magically" selects the correct x

Intermediate languages

Intermediate representations which can be represented textually as a program are often called **intermediate languages**.

Intermediate languages are similar to normal programming languages, but designed with different goals (*e.g.* simplicity, not conciseness).

Some intermediate languages are typed. This can help debugging the compiler, as the result of each phase can be type-checked.

Run time system

Implementing a high-level programming language usually means more than just writing a compiler.

A complete **run time system** must be written, to assist the execution of compiled programs by providing various services: memory management, threads, etc.

Automatic memory management

Most recent languages offer **automatic memory management**: the programmer allocates memory explicitly, but de-allocation is performed automatically.

The de-allocation of memory is usually performed by a part of the run time system called the **garbage collector** (GC).

Virtual machines

Instead of targeting a real processor, a compiler can target a virtual one, usually called a **virtual machine**.

The produced code is then interpreted by a program emulating the virtual machine.

Virtual machines advantages

Virtual machines are interesting for several reasons:

- the compiler can target a single (and usually high-level) architecture,
- the program can easily be monitored during execution, *e.g.* to prevent malicious behaviour, or provide debugging facilities,
- the distribution of compiled code is easier.

Virtual machines disadvantages

The main (only?) disadvantage of virtual machines is their speed: it is always slower to interpret a program in software than to execute it directly in hardware.

Dynamic compilation (a.k.a. JIT compilation)

To make virtual machines faster, **dynamic** (or **just-in-time**) **compilation** was invented.

The idea is simple: Instead of interpreting a piece of code, the virtual machine translates it to machine code, and hands it to the processor for execution.

This is usually faster than interpretation.

Summary

Compilers for high-level languages are more complex than the ones you've studied, since:

- they must translate high-level concepts like pattern-matching, closures, etc.
- they must be accompanied by a sophisticated run time system, and
- they should produced optimised code.

These will be the subjects of our study.

Real-world examples

The Scala compiler (v2.0)

The (new) Scala compiler is composed of approximately 13 phases:

- the first 10 are mostly simplification phases, which work on the AST and translate concepts like nested classes, closures, etc.
- the last 3 work on a hybrid IR called ICode.

The run time system is a standard JVM, since the compiler emits standard Java class files.

The OCaml compilers

There are two OCaml compilers: the first produces code for a virtual machine, the second produces machine code for several architectures.

The virtual machine, called ZAM_2, is a stack-based VM designed for the efficient interpretation of OCaml programs.

Two implementations of the ZAM_2 exist: a threaded-code interpreter, and a JIT compiler.

The OCaml compilers

The two compilers share:

- a common front-end, composed of the scanner, parser, type-checker and a first simplification phase,
- part of the run time system (mostly the GC).

The native compiler has a lot more phases, which handle problems like register allocation.