Introduction to program optimisation

Michel Schinz (based on Erik Stenman's slides)

Advanced Compiler Construction / 2006-05-19

Program optimisation

What is program optimisation?

The goal of program optimisation is to discover, at compilation time, information about the run-time behaviour of the program, and use that information to improve the generated code.

What *improving* means depends on the situation: often it implies reducing the execution time, but it can also imply reducing the size of the generated code, or the consumed memory, etc.

Correctness of optimisations

The most important feature of any optimisation is that it is **correct**, in the sense that it preserves the behaviour of the original program.

This implies in particular that if the original program would have failed during execution, the optimised one must also fail, and for the same reason – a property that is often forgotten.

Unattainable optimality

The term *optimisation* seems to imply that the resulting program is optimal.

It can be shown, however, that it is *not* possible to completely optimise a program, as this would make the halting problem solvable.

So optimisation is really about improving the generated code, not about making it optimal.

Anatomy of an optimisation

All optimisations can be seen as being composed of two phases:

1. an analysis phase, during which some part of the program is examined and properties are extracted,
2. a rewriting phase, during which the optimisation is applied by transforming the program, according to the result of the analysis.

Optimisation kinds

Two kinds of optimisations can be distinguished:

- **machine-independent optimisations**, which decrease the amount of work that the program has to perform – e.g. dead code elimination,
- **machine-dependent optimisations**, which take advantage of characteristics of the target machine – e.g. instruction scheduling.

Machine-independent optimisations examples

Machine-independent optimisations include:

- **constant folding**, which replaces constant expressions by their value,
- **common sub-expression elimination**, which avoids repeated evaluation of expressions,
- **dead-code elimination**, which eliminates code that will never be executed,
- etc.

Machine-dependent optimisations examples

Machine-dependent optimisations include:

- **instruction scheduling**, which rearranges instructions to avoid processor stalls,
- **register allocation**, which tries to use registers instead of memory as much as possible,
- **peephole optimisation**, which replaces given instruction sequences by faster alternatives,
- etc.

Optimisation scope

Optimisations can also be categorised according to their scope, that is the part of the program they analyse and transform:

- **local optimisations** consider basic blocks,
- **global optimisations** consider whole functions,
- **whole-program optimisations** consider the complete program.

Representing programs for optimisation

The representation used for the program plays a crucial role for optimisation. It must be at the right level of abstraction so that:

- the analysis is as easy as possible,
- no opportunities are lost – *e.g.* some common sub-expressions only appear after high-level constructs like array access have been translated to more basic instructions.

When to optimise?

Optimisation phases can be placed at various stages of the compilation process.

Machine-independent optimisations tend to be placed at the beginning, and work on high-level representations of the program (*e.g.* the AST).

Machine-dependent optimisations tend to be placed at the end, and work on low-level representations of the program (*e.g.* linear code).

Inlining

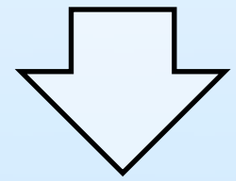
Inlining

Inlining (also called **inline expansion**) consists in replacing a call to a function with the body of that function – augmented with appropriate bindings for parameters.

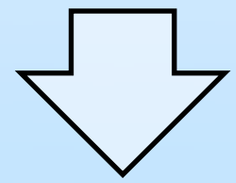
In other words, it consists in performing β-reduction – *i.e.* function application – during compilation.

Inlining example

```
(define + (lambda (x y) ($+ x y)))
(+ (+ 1 2) 3)
```



```
(+ (let ((x1 1) (y1 2)) ($+ x1 y1)) 3)
```



```
(let ((x2 (let ((x1 1) (y1 2)) ($+ x1 y1)))
      (y2 3))
  ($+ x2 y2))
```

Inlining and other optimisations

Inlining often opens the door to many other optimisations, as the inlined function can be specialised to its environment.

In our example, after inlining, the whole expression could be replaced by its value (6) using a series of well-known optimisations.

What should be inlined?

Inlining cannot be performed indiscriminately as this would result in code size explosion in most cases. Therefore, heuristics have to be used to decide when inlining should be performed.

These heuristics are generally based on the size of the function to inline or the “importance” of the call site. Also, functions which are called from a single location in the program can always be inlined, and the original version deleted.

Implementing inlining

Inlining is relatively straightforward to implement, and can be performed early in the compilation process. Two potential problems must be considered:

1. name capture, which can appear if the body of the inlined function is not α-renamed before being inlined,
2. non-termination, which can occur when recursive functions are inlined blindly.

Inlining requirement

In order to inline a function call, it must be possible to determine statically the function which will be invoked at run time.

As we have seen, this is generally impossible to do in object-oriented languages, because of the dynamic nature of method dispatch.

The same problem appears with higher-order functions, which are heavily used in functional languages.

Static inlining in object-oriented languages

Inlining in object-oriented languages can only be performed when the set of potential method bodies designated by a call is small – *e.g.* a singleton.

Computing these sets statically either requires an analysis of the whole program, or has to rely on specific characteristics of the language. In Java, for example, a call to a static method always refers to a single method body.

Dynamic inlining in object-oriented languages

Given the difficulty of statically computing the set of potential method bodies designated by a call, an option is to determine them dynamically.

Polymorphic inline caching does precisely that. As we have seen, this implies that it is possible to inline method bodies inside the specialised dispatching methods attached to call sites.

Inlining in functional languages

Inlining in functional languages can only be performed when the set of functional values which can flow to a given call site is small – typically a singleton.

An analysis called control-flow analysis (CFA) can be used to compute conservative approximations of these sets.

Summary

The goal of optimisations is to analyse the program and then transform it based on that analysis, so that it performs better in some respect.

Inlining is one example of optimisation. It consists in replacing a call to a known function by the body of that function. It is interesting in itself as it saves the cost of a function call, but also because it enables further optimisation.