Introduction to program optimisation

Michel Schinz – based on Erik Stenman's slides
Advanced compiler construction, 2008-05-02

Program optimisation

What is 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.

In this course, we will concentrate on the optimisation of execution time.

Correctness of optimisation

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.

Optimisation 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.

Optimisation 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** work on basic blocks,
- **global optimisations** work on whole functions (and not on the whole program as their name suggests),
- **whole-program optimisations** work on the complete program.

Program representation

The representation used for the program plays a crucial role for optimisation. It must be at the right level of abstraction to ensure 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 (or **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

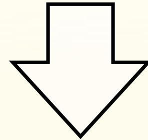
```
(car (cons 1 2))
```

Inlining example

```
(car ((cons 1 2)))
```

Inlining example

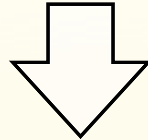
```
(car (cons 1 2))
```



```
(car (let ((fst 1) (snd 2))
       (vector fst snd)))
```

Inlining example

```
(car (cons 1 2))
```



```
(car (let ((fst 1) (snd 2))
       (vector fst snd)))
```

Inlining example

```
(let ((pair (let ((fst 1) (snd 2))
               (vector fst snd))))
  (vector-ref pair 0))
```

Inlining and other optimisations

In itself, inlining is already interesting as it saves the cost of function calls.

Moreover, 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 (1), using a series of well-known optimisations.

Inlining heuristics

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 that are called from a single location in the program can always be inlined, and the original version deleted.

Implementing inlining

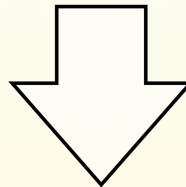
Inlining is relatively straightforward to implement, and can be performed early in the compilation process. Still, as we will see, a few pitfalls must be avoided!

Notice that most of these pitfalls can be avoided quite simply by choosing an appropriate representation for programs.

Pitfall: name capture

```
(define x 1)
(define succ (lambda (y) (+ y x)))
(define succ2 (lambda (x) (succ (succ x))))
```

incorrect inlining of `succ` in `succ2`



```
(define succ2 (lambda (x) (+ (+ x x) x)))
```

name capture

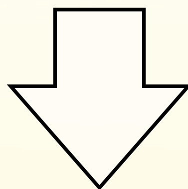
name capture

Solution: use unique names, or some equivalent (*e.g.* symbols)

Pitfall: side-effect duplication

```
(define print-and-ret (lambda (x) (print-int x) x))
(define twice (lambda (y) (+ y y)))
(define f (lambda (z) (twice (print-and-ret z))))
```

incorrect inlining of `twice` in `f`



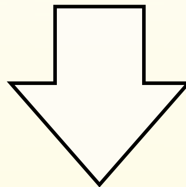
```
(define f (lambda (z)
            (+ (print-and-ret z)
               (print-and-ret z))))
```

Solution: bind actual parameters to variables (using a `let`) to ensure that they are evaluated once.

Pitfall: side-effect elimination

```
(define first (lambda (x y) x))
(define print-and-ret
  (lambda (z) (first z (print-int z))))
```

incorrect inlining of `first`
in `print-and-ret`

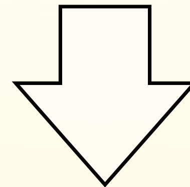


```
(define print-and-ret (lambda (z) z))
```

Solution: bind actual parameters to variables (using a `let`) to ensure that they are evaluated once.

Pitfall: recursive functions

```
(define fact
  (lambda (x)
    (if (= 0 x) 1 (* x (fact (- x 1))))))
```



correct inlining of `fact` in itself, but when should it end?

```
(define fact
  (lambda (x)
    (if (= 0 x) 1
        (* x (let ((x1 (- x 1)))
               (if (= 0 x1) 1
                   (* x1 (fact (- x1 1)))))))))
```

Solution: only inline recursive functions a limited number of times – possibly 0.

Inlining requirements

To be able to perform inlining at some point, it must be possible to determine statically the function that will be called.

This is easy in languages like C where most function calls designate a function using its name.

In object-oriented languages, this is much harder as most calls are method calls. As we have seen, it is generally not possible to know *statically* which actual method implementation corresponds to a given method call.

In functional languages, the extensive use of higher-order function causes the same problem.

Inlining in high-level languages

To perform inlining in object-oriented or functional languages, two techniques can be used:

1. A static analysis like **CFA** (**control-flow analysis**) can be used to compute, for every call site, a good approximation of the set of all possible targets of a call. If this set is a singleton – or, more generally, if it is small – inlining can be performed.
2. The actual targets of a call can be discovered at run time, and inlining can be performed then. This is the idea of the **polymorphic inline caching** technique we examined. It requires dynamic code generation.

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.