Code optimization

Advanced Compiler Construction
Michel Schinz – 2014–03–27

1

Optimizations

The goal of optimizations is to rewrite the program being compiled to a new program that is simultaneously:

- behaviorally equivalent to the original one,
- better in some respect – e.g. faster, smaller, more energy-efficient, etc.

Optimizations can be broadly split in two classes:

- **machine-independent optimizations**, like constant folding or dead code elimination, are high-level and do not depend on the target architecture,
- **machine-dependent optimizations**, like register allocation or instruction scheduling, are low-level and depend on the target architecture.

2

Rewriting optimizations

In this lesson, we will examine a simple set of machine-independent rewriting optimizations.
Most of them are relatively simple rewrite rules that transform the source program to a shorter, equivalent one.

3

IRs and optimizations

4

The importance of IRs

The intermediate representation (IR) on which rewriting optimizations are performed can have a dramatic impact on their ease of implementation.

A rewriting optimization generally works in two steps:

1. the program is analyzed to find rewrite opportunities,
2. the program is rewritten based on the opportunities identified in the first step.

A good IR should make both steps as easy as possible. The following examples illustrate the importance of using the right IR.

Case 1: constant propagation

To illustrate the impact of IR on the analysis step, consider the following program fragment in some imaginary IR:

```
x←7
…
```

Is it legal to perform constant propagation and blindly replace all later occurrences of x by 7? It depends on the IR:

- If multiple assignments to the same variable are allowed, then additional (data-flow) analyses are required to answer the question, as x might be re-assigned later.
- However, if multiple assignments to the same variable are prohibited, then yes, all occurrences of x can be unconditionally replaced by 7!

Other simple optimizations

Apart from constant propagation, many simple optimizations are made hard by the presence of multiple assignments to a single variable:

- **common-subexpression elimination**, which consists in avoiding the repeated evaluation of expressions,
- (simple) **dead code elimination**, which consists in removing assignments to variables whose value is not used later,
- etc.

In all cases, analyses are required to distinguish the various "versions" of a variable that appear in the program.

Conclusion: a good IR should not allow multiple assignments to a variable!

Case 2: inlining

Inlining (or **in-line expansion**) consists in replacing a call to a function by a copy of the body of that function, with parameters replaced by the actual arguments. It is a very important compiler optimization, as it often opens the door to further optimizations.

Some aspects of the intermediate representation can have an important impact on the implementation of inlining. To illustrate this, let us examine some problems that can occur when performing inlining directly on the AST – a choice that might seem reasonable at first sight.

Naïve inlining: problem #1

```
(def print/ret (fun (x) (int-print x) x))
(def twice (fun (y) (+ y y)))
(def f (fun (z) (twice (print/ret z))))
```

incorrect inlining of `twice` in `f`

```
(def f (fun (z)
         (+ (print/ret z)
            (print/ret z))))
```

z is printed twice!

Possible solution: bind actual parameters to variables (using a `let`) to ensure that they are evaluated *at most* once.

Naïve inlining: problem #2

```
(def first (fun (x y) x))
(def print/ret
  (fun (z) (first z (int-print z))))
```

incorrect inlining of `first` in `print/ret`

```
(def print/ret (fun (z) z))
```

z isn't printed at all!

Possible solution: bind actual parameters to variables (using a `let`) to ensure that they are evaluated *at least* once.

Easy inlining

The two pitfalls presented earlier can be avoided by bindings actual arguments to variables (using a `let`) before using them in the body of the inlined function.
However, a properly-designed IR can also avoid the problems altogether by ensuring that actual parameters are always atoms, i.e. variables or constants.
Conclusion: a good IR should only allow atomic arguments to functions.

IR comparison

Using the two very basic criterions we identified, we can evaluate the various classes of IRs we have seen:

- standard RTL/CFG is bad in that its variables are mutable; however, it allows only atoms as function arguments, which is good,
- RTL/CFG in SSA form, CPS/L_3 and similar functional IRs are good in that their variables are immutable, and they only allow atoms as function arguments.

Simple CPS/L_3 optimizations

Rewriting optimizations

The rewriting optimizations for CPS/L_3 are specified as a set of rewriting rules of the form $T \rightsquigarrow_{opt} T'$. These rules rewrite a CPS/L_3 term T to an equivalent – but hopefully more efficient – term T′.

Optimization rules, like desugaring rules, can only be applied in specific contexts, designated as C_{opt}.

Using optimization rewrite rules and contexts, the optimization relation mapping a CPS/L_3 program to its optimized form can be defined as usual:

$C_{opt}[T] \Rightarrow_{opt} C_{opt}[T']$ if $T \rightsquigarrow_{opt} T'$

Optimization contexts

The optimization contexts for CPS/L_3 are generated by the following grammar:

C_{opt} ::= □
| (let$_l$ ((n l)) C_{opt})
| (let$_p$ ((n (p n_1 …))) C_{opt})
| (let$_c$ ((c_1 e_1) … (c_i (cnt ($n_{i,1}$ …) C_{opt})) … (c_k e_k)) e)
| (let$_c$ ((c_1 e_1) … C_{opt})
| (let$_f$ ((f_1 e_1) … (f_i (fun ($n_{i,1}$ …) C_{opt})) … (f_k e_k)) e)
| (let$_f$ ((f_1 e_1) …) C_{opt})

(Non-)shrinking rules

We can distinguish two classes of rewriting rules:

1. **shrinking rules** rewrite a term to an equivalent but smaller one,
2. **non-shrinking rules** rewrite a term to an equivalent but potentially larger one.

Shrinking rules can be applied at will, possibly until the term is fully reduced, while non-shrinking rules cannot, as they would result in infinite expansion. Heuristics must be used to decide when to apply non-shrinking rules.

Except for (non-linear) inlining, all optimizations we will see are shrinking.

Dead code elimination

Dead code elimination removes all code that neither performs side effects nor produces a value used later.

$(\texttt{let}_l\ ((n\ l))\ e)$

$\rightsquigarrow_{opt}$ e [*if* n *is not free in* e]

$(\texttt{let}_p\ ((n\ (p\ n_1 \ldots)))\ e)$

$\rightsquigarrow_{opt}$ e [*if* n *is not free in* e *and* p $\notin$ { `char-read`, `char-print`, `/`, `%`, `block-set!` }]

$(\texttt{let}_f\ ((n_1\ f_1)\ \ldots\ (n_i\ f_i)\ \ldots\ (n_k\ f_k))\ e)$

$\rightsquigarrow_{opt}$ $(\texttt{let}_f\ ((n_1\ f_1)\ \ldots\ (n_k\ f_k))\ e)$

[*if* n_i *is not free in* $\{f_1, \ldots, f_{i-1}, f_{i+1}, \ldots f_k, e\}$]

The rule for continuations is similar to the one for functions.

Dead code elimination

The rewriting rules to eliminate dead functions does not eliminate dead but mutually-recursive functions.

To handle them, one must start from the main expression of the program, and identify all functions transitively reachable from it. All unreachable functions are dead.

The rule for continuations has the same problem, which admits a similar solution. The only difference is that continuations being local to functions, the reachability analysis can start in function bodies.

CSE

Common subexpression elimination (**CSE**) avoids the repeated evaluation of a single expression.

$(\texttt{let}_l\ ((n_1\ l))\ C_{opt}[(\texttt{let}_l\ ((n_2\ l))\ e)])$

$\rightsquigarrow_{opt}$ $(\texttt{let}_l\ ((n_1\ l))\ C_{opt}[e\{n_2 \rightarrow n_1\}])$

$(\texttt{let}_p\ ((n_1\ (+\ a_1\ a_2)))$

$C_{opt}[(\texttt{let}_p\ ((n_2\ (+\ a_1\ a_2)))\ e)])$

$\rightsquigarrow_{opt}$ $(\texttt{let}_p\ ((n_1\ (+\ a_1\ a_2)))\ C_{opt}[e\{n_2 \rightarrow n_1\}])$

etc.

η-reduction

Variants of the standard η-reduction can be performed to remove redundant definitions.

$(\texttt{let}_c\ ((c_1\ e_1)\ \ldots$

$(c_i\ (\texttt{cnt}\ (a_1 \ldots)\ (\texttt{app}_c\ d\ a_1 \ldots)))\ \ldots$

$(c_k\ e_k))$

$e)$

$\rightsquigarrow_{opt}$ $(\texttt{let}_c\ ((c_1\ e_1\{c_i \rightarrow d\})\ \ldots\ (c_k\ e_k\{c_i \rightarrow d\}))\ e\{c_i \rightarrow d\})$

$(\texttt{let}_f\ ((n_1\ f_1)\ \ldots$

$(n_i\ (\texttt{fun}\ (c\ a_1 \ldots)\ (\texttt{app}_f\ g\ c\ a_1 \ldots))\ \ldots$

$(n_k\ f_k))$

$e)$

$\rightsquigarrow_{opt}$ $(\texttt{let}_f\ ((n_1\ f_1\{n_i \rightarrow g\})\ \ldots\ (n_k\ f_k\{n_i \rightarrow g\}))\ e\{n_i \rightarrow g\})$

Constant folding (1)

Constant folding consists in replacing a constant expression by its value. Example for addition:

(let$_l$ ((n$_1$ l$_1$))
  C$_{opt}$[(let$_l$ ((n$_2$ l$_2$))
    C′$_{opt}$[(let$_p$ ((n$_3$ (+ n$_1$ n$_2$))) e)])])
  $\rightsquigarrow_{opt}$ (let$_l$ ((n$_1$ l$_1$))
      C$_{opt}$[(let$_l$ ((n$_2$ l$_2$))
        C′$_{opt}$[(let$_l$ ((n$_3$ l$_1$+l$_2$)) e)])])

Similar rules exist for other arithmetic primitives.
Constant expressions that would result in a run-time error – e.g. a division by zero – must not be rewritten!

Constant folding (2)

Primitives appearing in conditional expressions are also amendable to constant folding, for example:

(if (= n n) c$_t$ c$_f$)
  $\rightsquigarrow_{opt}$ (app$_c$ c$_t$)
(if (!= n n) c$_t$ c$_f$)
  $\rightsquigarrow_{opt}$ (app$_c$ c$_f$)
(let$_l$ ((n$_1$ l$_1$))
  C$_{opt}$[(let$_l$ ((n$_2$ l$_2$))
      C′$_{opt}$[(if (= n$_1$ n$_2$) c$_t$ c$_f$)])])
  $\rightsquigarrow_{opt}$ (let$_l$ ((n$_1$ l$_1$))
      C$_{opt}$[(let$_l$ ((n$_2$ l$_2$))
        C′$_{opt}$[(app$_c$ c$_t$)])]) [*if* l$_1$ = l$_2$]

etc.

Neutral/absorbing elements

Uses of neutral and absorbing elements of arithmetic primitives can be simplified. For multiplication, this is expressed by the following rules:

(let$_l$ ((n$_1$ 0)) C$_{opt}$[(let$_p$ ((n (* n$_1$ n$_2$))) e)])
  $\rightsquigarrow_{opt}$ (let$_l$ ((n$_1$ 0)) C$_{opt}$[e{n→n$_1$}])
(let$_l$ ((n$_2$ 0)) C$_{opt}$[(let$_p$ ((n (* n$_1$ n$_2$))) e)])
  $\rightsquigarrow_{opt}$ (let$_l$ ((n$_2$ 0)) C$_{opt}$[e{n→n$_2$}])
(let$_l$ ((n$_1$ 1)) C$_{opt}$[(let$_p$ ((n (* n$_1$ n$_2$))) e)])
  $\rightsquigarrow_{opt}$ (let$_l$ ((n$_1$ 1)) C$_{opt}$[e{n→n$_2$}])
(letl ((n$_2$ 1)) C$_{opt}$[(let$_p$ ((n (* n$_1$ n$_2$))) e)])
  $\rightsquigarrow_{opt}$ (let$_l$ ((n$_2$ 1)) C$_{opt}$[e{n→n$_1$}])

Similar rules exist for other arithmetic operators.

Block primitives

Block primitives are harder to optimize, because block elements can be modified.
However, some blocks used by the compiler, e.g. to implement closures, are known to be constant once initialized. This makes rewritings like the following possible:

(let$_p$ ((b (block-alloc-k s)))
  C$_{opt}$[(let$_p$ ((t (block-set! b i v)))
    C′$_{opt}$[(let$_p$ ((n (block-get b i))) e)])])
  $\rightsquigarrow_{opt}$ (let$_p$ ((b (block-alloc-k s)))
      C$_{opt}$[(let$_p$ ((t (block-set! b i v)))
        C′$_{opt}$[e{n→v}])])

[*when tag* k *identifies a block that is not modified after initialization, e.g. a closure block*]

Exercise

CPS/L_3 contains the following block primitives:

- `block-alloc-`n size
- `block-tag` block
- `block-size` block
- `block-get` block index
- `block-set!` block index value

Informally describe three rewriting optimizations that could be performed on these primitives, and in which conditions they could be performed.

CPS/L_3 inlining

Shrinking inlining

A non-recursive continuation or function that is applied exactly once – i.e. used in a linear fashion – can always be inlined without making the code grow:

(let$_f$ ((f_1 e_1) … (f_i (fun (c_i $n_{i,1}$ …) e_i)) … (f_k e_k))
  C_{opt}[(app$_f$ f_i c m_1 …)])
  $\rightsquigarrow_{opt}$ (let$_f$ ((f_1 e_1) … (f_k e_k))
    C_{opt}[e_i{c_i→c}{$n_{i,1}$→m_1}…])
  [*when* f_i *is not free in* C_{opt}, e_1, …, e_n]

The rule for continuations is similar.

General inlining

Non-linear inlining can also be performed trivially in CPS/L_3, either for continuation or for functions (illustrated here):

(let$_f$ (… (f_i (fun (c_i $n_{i,1}$ …) e_i)) …)
  C_{opt}[(app$_f$ f_i c m_1 …)])
  $\rightsquigarrow_{opt}$ (let$_f$ (… (f_i (fun (c_i $n_{i,1}$ …) e_i)) …)
    C_{opt}[e_i{c_i→c}{$n_{i,1}$→m_1}…])

(To preserve the uniqueness of names, fresh versions of bound names should be created during inlining.)

The problem of these rules is that they are not shrinking and rewriting does not even terminate with recursive continuations or functions.

Inlining heuristics

Since non-shrinking inlining cannot be performed indiscriminately, heuristics are used to decide whether a candidate function should be inlined at a given call site. These heuristics typically combine several factors, like:

- the size of the candidate function – smaller ones should be inlined more eagerly than bigger ones,
- the number of times the candidate is called in the whole program – a function called only a few times should be inlined,

(*continued on next slide*)

Inlining heuristics (2)

- the nature of the candidate – not much gain can be expected from the inlining of a recursive function,
- the kind of arguments passed to the candidate, and/or the way these are used in the candidate – constant arguments could lead to further reductions in the inlined candidate, especially if it combines them with other constants,
- etc.

Exercise

Imagine an imperative intermediate language equipped with a `return` statement to return from the current function to its caller.

1. Describe the problem that would appear when inlining a function containing such a `return` statement.
2. Explain how a return statement could be encoded in CPS/L_3 and why such an encoding would not suffer from the above problem.

CPS/L_3 "contification"

Contification

Contification is the name generally given to an optimization that transforms functions into (local) continuations.
When applicable, this transformation is interesting because it transforms expensive functions – compiled as closures – to inexpensive continuations – compiled as code blocks.

Contification example

Contification can for example transform the `loop` function in the L_3 example below to a local continuation, leading to efficient compiled code.

```
(def fact
  (fun (x)
    (rec loop ((i 1) (r 1))
      (if (> i x)
          r
          (loop (+ i 1) (* r i))))))
```


Non-recursive contification

A non-recursive function f is contifiable if and only if it always returns to the same location, because then it doesn't need a return continuation.
In CPS/L_3, this condition is equivalent to requiring that f is only used in $\texttt{app}_f$ nodes, in function position, and always passed the same return continuation.

Non-recursive contification

The contification of the non-recursive function f is given by:

$$\texttt{(let}_f\ \texttt{((}f\ \texttt{(fun}\ \texttt{(}c\ a_1 \ldots\texttt{)}\ e\texttt{)))}$$
$$C_{opt}[C'_{opt}[\texttt{(app}_f\ f\ c_0\ n_{1,1} \ldots\texttt{)}, \texttt{(app}_f\ f\ c_0\ n_{2,1}, \ldots\texttt{)}, \ldots]]\texttt{)}$$
$$\rightsquigarrow_{opt} C_{opt}[\texttt{(let}_c\ \texttt{((}\underline{\texttt{m}}\ \texttt{(cnt}\ \texttt{(}a_1 \ldots\texttt{)}\ e\{c \rightarrow c_0\}\texttt{)))}$$
$$C'_{opt}[\texttt{(app}_c\ \texttt{m}\ n_{1,1} \ldots\texttt{)}, \texttt{(app}_c\ \texttt{m}\ n_{2,1} \ldots\texttt{)}, \ldots]\texttt{)}]$$

where f does not appear free in either C_{opt} or C'_{opt}, and C'_{opt} is the smallest (multi-hole) context enclosing all applications of f. It ensures that the scope of `m` is as small as possible.

Recursive contification

A set of mutually-recursive functions $F = \{ f_1, \dots, f_n \}$ can be contified together – which we write Contifiable(F) – if and only if every function in F is always used in one of the following two ways:

1. applied to a common return continuation c_0, or
2. called in tail position by a function in F.

Intuitively, this ensures that all functions in F eventually return through the common continuation c_0.

Example

As an example, the functions `even` and `odd` in the CPS/L_3 translation of the following L_3 term are contifiable:

```
(letrec
   ((even (fun (x)
            (if (= 0 x) #t (odd (- x 1)))))
    (odd (fun (x)
            (if (= 0 x) #f (even (- x 1))))))
  (even 12))
```

Contifiable(F = {`even`, `odd`}) is satisfied since:

1. the single use of `odd` is a tail call from `even` ∈ F,
2. `even` is tail-called from `odd` ∈ F and called with the continuation of the `letrec` statement – the common return continuation c_0 for this example.

Recursive contification

Given a set of mutually-recursive functions

```
(let_f ((f_1 e_1) (f_2 e_2) … (f_n e_n))
  e)
```

the condition Contifiable(F) for some $F \subseteq \{ f_1, \dots, f_n \}$ can only be true if all the non tail calls to functions in F appear either:

- in the term e, or
- in the body of exactly one function $f_i \notin F$.

Therefore, two separate rewriting rules must be defined, one per case.

Recursive contification #1

When all non tail calls to functions in $F = \{ f_1, \dots, f_i \}$ appear in the body of the `let_f`, and Contifiable(F) holds, contification is performed by the following rewriting:

```
(let_f ((f_1 (fun (c_1 a_1,1 …) e_1))  … (f_n …))
  C_opt[e])
  ⇝_opt (let_f ((f_i+1 (fun (c_i+1 a_i+1,1 …) e_i+1))…(f_n …))
          C_opt[(let_c ((m1 (cnt (a_1,1 …)
                                 e_1*{c_1→c_0})) …)
                  e*)])
```

where $f_1, \dots, f_i$ do not appear free in C_{opt} and e is minimal.

Note: the term t^* is t with all applications of contified functions transformed to continuation applications.

Recursive contification #2

When all non tail calls to functions in $F = \{ f_1, \ldots, f_i \}$ appear in the body of the function f_n, and Contifiable(F) holds, contification is performed by the following rewriting:

$$
\begin{array}{l}
\texttt{(let}_{\texttt{f}}\ \texttt{((}f_1\ \texttt{(fun (}c_1\ a_{1,1} \ldots\texttt{)}\ e_1\texttt{))}\ \ldots \\
\qquad\qquad \texttt{(}f_n\ \texttt{(fun (}c_n\ a_{n,1} \ldots\texttt{)}\ C_{opt}[e_n]\texttt{)))}\ e\texttt{)} \\
\quad \rightsquigarrow_{opt} \texttt{(let}_{\texttt{f}}\ \texttt{((}f_{i+1}\ \texttt{(fun (}c_{i+1}\ a_{i+1,1} \ldots\texttt{)}\ e_{i+1}\texttt{))}\ \ldots \\
\qquad\qquad\qquad \texttt{(}f_n\ \texttt{(fun (}c_n\ a_{n,1} \ldots\texttt{)} \\
\qquad\qquad\qquad\qquad C_{opt}[\texttt{(let}_{\texttt{c}}\ \texttt{((}\underline{\texttt{m1}}\ \texttt{(cnt (}a_{1,1} \ldots\texttt{)} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad e_1^*\{c_1 \rightarrow c_0\}\texttt{))} \\
\qquad\qquad\qquad\qquad\qquad\qquad \ldots\texttt{)} \\
\qquad\qquad\qquad\qquad\qquad e_n^*\texttt{)}]\texttt{)))}\ e\texttt{)}
\end{array}
$$

where $f_1, \ldots, f_i$ do not appear free in C_{opt} and e_n is minimal.