Code optimization

Advanced Compiler Construction
Michel Schinz – 2024-03-28

Optimization

Goal: rewrite the program to a new one that is:

- 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** are high-level and do not depend on the target architecture,
- **machine-dependent optimizations** are low-level and depend on the target architecture.

This lesson: machine-independent, rewriting optimizations.

IRs and optimizations

The importance of IRs

Intermediate representations (IRs) have a dramatic impact on optimizations, which generally work in two steps:

1. the program is analyzed to find optimization opportunities,
2. the program is rewritten based on the analysis.

The IR should make both steps as easy as possible.

Case 1: constant propagation

Consider the following program fragment in some imaginary IR:

```
x ← 7
...
```

Question: can all occurrences of x be replaced by 7?

Answer: it depends on the IR:

- if it allows multiple assignments, no (further data-flow analyses are required),
- if it disallows multiple assignment, yes!

Other simple optimizations

Multiple assignments make most simple optimizations hard:

- *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.

Common problem: 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 replaces a call to a function by a copy of the body of that function, with parameters replaced by the actual arguments.
The IR used also has a dramatic impact on it, as we can see if we try to do inlining on the AST – which might look sensible at first.

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

Common solution:

bind actual arguments to variables before using them in the body of the inlined function.

However:

the IR can also avoid the problem by ensuring that actual parameters are always atoms (variables/constants).

Conclusion:

a good IR should only allow atomic arguments to functions.

IR comparison

Conclusion:

- standard RTL/CFG is:
  - bad as its variables are mutable, but
  - good as it allows only atoms as function arguments,
- RTL/CFG in SSA form and CPS/L_3 are:
  - good as their variables are immutable,
  - good as 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'.

(Non-)shrinking rules

We can distinguish two classes of rewriting rules:

1. **shrinking rules** rewrite a term to an equivalent but smaller one, and can be applied at will,
2. **non-shrinking rules** rewrite a term to an equivalent but potentially larger one, and must be applied carefully.

Except for inlining, all optimizations we will see are shrinking.

Optimization contexts

Rewriting rules can only be applied in specific locations. For example, it would be incorrect to try to rewrite the parameter list of a function.
We express this constraint by specifying all the **contexts** in which it is valid to perform a rewrite, where a context is a term with a single **hole** denoted by `□`.

The hole of a context C can be plugged with a term T, an operation written as C[T].
For example, if C is `(if □ ct cf)`, then C[`(= x y)`] is

`(if (= x y) ct cf)`.

Optimization contexts

C_{opt} ::= $\square$

| (let$_p$ ((n (p a$_1$ …))) 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})

Optimization relation

By combining the optimization rewriting rules – presented later – and the optimization contexts, it is possible to specify the optimization relation $\Rightarrow_{opt}$ that rewrites a term to an optimized version:

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

Dead code elimination

$(\texttt{let}_p\ ((n\ (p\ a_1\ \ldots)))\ e)$

$\quad \rightsquigarrow_{opt}\ e$

[*when* n *is not free in* e *and* $p \notin$ { `byte-read`, `byte-write`, `block-set!` }]

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

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

[*when* 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

Limitation:

Does not eliminate dead, mutually-recursive functions.

Solution:

- start from the main expression of the program, and
- identify all functions transitively reachable from it.

All unreachable functions are dead.

CSE

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

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

etc.

CSE

Limitation:

Some opportunities are missed because of scoping.

Example:

Common subexpression `(+ y z)` is not optimized:

```
(letc ((c1 (cnt ()
              (letp ((x1 (+ y z)))
                …)))
       (c2 (cnt ()
              (letp ((x2 (+ y z)))
                …))))
  …)
```

η-reduction

```
(letc ((c1 e1) ...
        (ci (cnt (n1 ...) (appc d n1 ...))) ...
        (ck ek))
  e)
  ⇝opt (letc ((c1 e1{ci→d}) ... (ck ek{ci→d})) e{ci→d})

(letf ((n1 f1) ...
        (ni (fun (c m1 ...) (appf g c m1 ...)) ...
        (nk fk))
  e)
  ⇝opt (letf ((n1 f1{ni→g}) ... (nk fk{ni→g})) e{ni→g})
```

[*when* $g \notin \{m_1, \ldots\}$]

Constant folding (1)

$(\mathtt{let}_p\ ((n\ (+\ l_1\ l_2)))\ e)$

$\quad \rightsquigarrow_{opt} e\{n \rightarrow (l_1 + l_2)\}$

[*when* l_1 *and* l_2 *are integer literals*]

$(\mathtt{let}_p\ ((n\ (-\ l_1\ l_2)))\ e)$

$\quad \rightsquigarrow_{opt} e\{n \rightarrow (l_1 - l_2)\}$

[*when* l_1 *and* l_2 *are integer literals*]

$(\mathtt{let}_p\ ((n\ (*\ l_1\ l_2)))\ e)$

$\quad \rightsquigarrow_{opt} e\{n \rightarrow (l_1 \times l_2)\}$

[*when* l_1 *and* l_2 *are integer literals*]

etc.

Constant folding (2)

$(\texttt{if}\ (=\ a\ a)\ c_t\ c_f)$

$\rightsquigarrow_{opt}\ (\texttt{app}_c\ c_t)$

$(\texttt{if}\ (<\ a\ a)\ c_t\ c_f)$

$\rightsquigarrow_{opt}\ (\texttt{app}_c\ c_f)$

etc.

Neutral/absorbing elements

```
(letₚ ((n (* 1 a))) e)
```
$\rightsquigarrow_{opt}$ e{n→a}

```
(letₚ ((n (* a 1))) e)
```
$\rightsquigarrow_{opt}$ e{n→a}

```
(letₚ ((n (* 0 a))) e)
```
$\rightsquigarrow_{opt}$ e{n→0}

```
(letₚ ((n (* a 0))) e)
```
$\rightsquigarrow_{opt}$ e{n→0}

etc.

Block primitives

```
(let_p ((b (block-alloc t s)))
  C_opt[(let_p ((u (block-set! b i a)))
    C'_opt[(let_p ((n (block-get b i))) e)])])
  ⇝_opt (let_p ((b (block-alloc t s)))
          C_opt[(let_p ((u (block-set! b i a)))
            C'_opt[e{n→a}])])
```

[*when tag* t *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` tag 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

(Non-)shrinking inlining

We can distinguish two kinds of inlining:

1. **shrinking inlining**, for functions/continuations that are applied exactly once,
2. **non-shrinking inlining**, for other functions/continuations.

Shrinking inlining can be applied at will, non-shrinking cannot.

Shrinking Inlining

$$\texttt{(let}_f\ \texttt{((}f_1\ e_1\texttt{)}\ \ldots\ \texttt{(}f_i\ \texttt{(fun}\ \texttt{(}c_i\ n_{i,1}\ \ldots\texttt{)}\ e_i\texttt{))}\ \ldots\ \texttt{(}f_k\ e_k\texttt{))}$$

$$C_{opt}[\texttt{(app}_f\ f_i\ c\ m_1\ \ldots\texttt{)}]\texttt{)}$$

$$\rightsquigarrow_{opt}\ \texttt{(let}_f\ \texttt{((}f_1\ e_1\texttt{)}\ \ldots\ \texttt{(}f_k\ e_k\texttt{))}$$

$$C_{opt}[e_i\{c_i \rightarrow c\}\{n_{i,1} \rightarrow m_1\}\ldots]\texttt{)}$$

[*when* f_i *is not free in* C_{opt}, e_1, …, e_n]

Similar rules exist to do the inlining inside of the body of one of the functions.

Non-shrinking Inlining

In non-shrinking inlining, fresh versions of bound names should be created to preserve their global uniqueness:

$$
\begin{array}{l}
\texttt{(let}_{\texttt{f}}\ \texttt{(}\ldots\ \texttt{(}f_i\ \texttt{(fun}\ \texttt{(}c_i\ n_{i,1}\ \ldots\texttt{)}\ e_i\texttt{))}\ \ldots\texttt{)} \\
\quad C_{opt}[\texttt{(app}_{\texttt{f}}\ f_i\ c\ m_1\ \ldots\texttt{)}]\texttt{)} \\
\quad \rightsquigarrow_{opt}\ \texttt{(let}_{\texttt{f}}\ \texttt{(}\ldots\ \texttt{(}f_i\ \texttt{(fun}\ \texttt{(}c_i\ n_{i,1}\ \ldots\texttt{)}\ e_i\texttt{))}\ \ldots\texttt{)} \\
\qquad\qquad C_{opt}[e_i\{c_i \rightarrow c\}\{n_{i,1} \rightarrow m_1\}\ldots]\texttt{)}
\end{array}
$$

Similar rules exist to do the inlining inside of the body of one of the functions.

Inlining heuristics (1)

Heuristics must be used to decide when to perform non-shriking inlining.
They 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: transforms functions into continuations.
Interesting optimization as it transforms functions, which are expensive (closures) into continuations, which are cheap.

Contification example

Example: the `loop` function in the L_3 example below can be contified, leading to efficient compiled code.

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

Contifiability

A CPS/L_3 function is contifiable if and only if it always returns to the same location – because then it does not need a return continuation.

- Non-recursive case: true iff that function is only used in app_f nodes, in function position, and always passed the same return continuation.
- Recursive case: slightly more involved – see later.

Non-recursive contification

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

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

where:

- f does not appear free in C_{opt} or C'_{opt},
- C'_{opt} is the smallest (multi-hole) context enclosing all applications of f,
- c_0 is the (single) return continuation that is passed to function f.

Recursive contifiability

A set of mutually-recursive functions $F = \{ f_1, \ldots, f_n \}$ is contifiable – which we write Cnt(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, 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.

Example

As an example, 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))
```

Cnt(F = {**even**, **odd**}) is satisfied since:

- the single use of **odd** is a tail call from **even** ∈ F,
- **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 ((f1 e1) (f2 e2) … (fn en))
   e)
```

the condition Cnt(F) for some $F \subseteq \{ f_1, \ldots, f_n \}$ 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

Case 1: all non tail calls to functions in F = { f_1, …, f_i } appear in the body of the `let`$_f$, and Cnt(F) holds:

```
(letf ((f1 (fun (c1 a1,1 …) e1)) … (fn …))
   Copt[e])
   ⇝opt (letf ((fi+1 (fun (ci+1 ai+1,1 …) ei+1))…(fn …))
          Copt[(letc ((m1 (cnt (a1,1 …)
                                e1*{c1→c0})) …)
                e*)])
```

where f_1, …, 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

Case 2: all non tail calls to functions in F = { f_1, …, f_i } appear in the body of the function f_n, and Cnt(F) holds:

```
(letf ((f₁ (fun (c₁ a₁,₁ …) e₁)) …
       (fₙ (fun (cₙ aₙ,₁ …) Copt[eₙ]))) e)
  ⇝opt (letf ((fᵢ₊₁ (fun (cᵢ₊₁ aᵢ₊₁,₁ …) eᵢ₊₁)) …
              (fₙ (fun (cₙ aₙ,₁ …)
                    Copt[(letc ((m₁ (cnt (a₁,₁ …)
                                          e₁*{c₁→c₀}))
                                …)
                           eₙ*)]))) e)
```

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

Contifiable subsets

Given a `let`$_f$ term defining a set of functions $F = \{ f_1, \ldots, f_n \}$, the subsets of F of potentially contifiable functions are obtained by:

1. building the tail-call graph of its functions, identifying the functions that call each-other in tail position,
2. extracting the strongly-connected components of that graph.

A given set of strongly-connected functions $F_i \subseteq F$ is then either contifiable together, i.e. $Cnt(F_i)$, or not contifiable at all – i.e. none of its subsets of functions are contifiable.