

# Code optimization

Advanced Compiler Construction  
Michel Schinz – 2021-04-01

## 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<sub>3</sub> are:
  - good as their variables are immutable,
  - good as they only allow atoms as function arguments.

## Simple CPS/L<sub>3</sub> optimizations

## Rewriting optimizations

The rewriting optimizations for CPS/L<sub>3</sub> are specified as a set of rewriting rules of the form  $T \rightarrow_{\text{opt}} T'$ .

These rules rewrite a CPS/L<sub>3</sub> 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  $\square$ .

The hole of a context  $C$  can be plugged with a term  $T$ , an operation written as  $C[T]$ .

For example, if  $C$  is  $(\text{if } \square \text{ ct cf})$ , then  $C[(= x y)]$  is

$(\text{if } (= x y) \text{ ct cf})$ .

## Optimization contexts

$C_{\text{opt}} ::= \square$

$| (\text{let}_p ((n (p \ n_1 \dots))) \ C_{\text{opt}})$   
 $| (\text{let}_c ((c_1 \ e_1) \dots (c_i (\text{cnt } (n_{i,1} \dots) \ C_{\text{opt}})) \dots (c_k \ e_k)) \ e)$   
 $| (\text{let}_c ((c_1 \ e_1) \dots) \ C_{\text{opt}})$   
 $| (\text{let}_f ((f_1 \ e_1) \dots (f_i (\text{fun } (n_{i,1} \dots) \ C_{\text{opt}})) \dots (f_k \ e_k)) \ e)$   
 $| (\text{let}_f ((f_1 \ e_1) \dots) \ C_{\text{opt}})$

## Optimization relation

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

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

## Dead code elimination

$(\text{let}_p ((n (p v_1 \dots))) e)$   
 $\twoheadrightarrow_{\text{opt}} e$   
*[when  $n$  is not free in  $e$  and  $p \notin \{\text{byte-read}, \text{byte-write}, \text{block-set!}\}$ ]*

$(\text{let}_f ((n_1 f_1) \dots (n_i f_i) \dots (n_k f_k)) e)$   
 $\twoheadrightarrow_{\text{opt}} (\text{let}_f ((n_1 f_1) \dots (n_k f_k)) e)$   
*[when  $n_i$  is not free in  $\{f_1, \dots, f_{i-1}, f_{i+1}, \dots, f_k, e\}$ ]*

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

$(\text{let}_p ((n_1 (+ v_1 v_2)))$   
 $C_{\text{opt}}[(\text{let}_p ((n_2 (+ v_1 v_2))) e)])$   
 $\twoheadrightarrow_{\text{opt}} (\text{let}_p ((n_1 (+ v_1 v_2))) C_{\text{opt}}[e\{n_2 \rightarrow n_1\}])$

$(\text{let}_p ((n_1 (- v_1 v_2)))$   
 $C_{\text{opt}}[(\text{let}_p ((n_2 (- v_1 v_2))) e)])$   
 $\twoheadrightarrow_{\text{opt}} (\text{let}_p ((n_1 (- v_1 v_2))) C_{\text{opt}}[e\{n_2 \rightarrow n_1\}])$

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))
      ...)))
  ...))
```

21

## 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})
```

22

## Constant folding (1)

```
(letp ((n (+ l1 l2))) e)
→opt e{n→(l1+l2)}
[when l1 and l2 are integer literals]
```

```
(letp ((n (- l1 l2))) e)
→opt e{n→(l1-l2)}
[when l1 and l2 are integer literals]
```

```
(letp ((n (* l1 l2))) e)
→opt e{n→(l1×l2)}
[when l1 and l2 are integer literals]
```

etc.

23

## Constant folding (2)

```
(if (= v v) ct cf)
→opt (appc ct)
```

```
(if (= l1 l2) ct cf)
→opt (appc ct)
[when l1 and l2 are literals and l1 = l2]
```

etc.

24

## Neutral/absorbing elements

```
(letp ((n (* 1 v))) e)
```

$\mapsto_{\text{opt}}$   $e\{n \rightarrow v\}$

```
(letp ((n (* v 1))) e)
```

$\mapsto_{\text{opt}}$   $e\{n \rightarrow v\}$

```
(letp ((n (* 0 v))) e)
```

$\mapsto_{\text{opt}}$   $e\{n \rightarrow 0\}$

```
(letp ((n (* v 0))) e)
```

$\mapsto_{\text{opt}}$   $e\{n \rightarrow 0\}$

etc.

26

## Block primitives

```
(letp ((b (block-alloc-k)))
```

$C_{\text{opt}}[(\text{let}_p ((t (\text{block-set! } b \ v))))$

$C'_{\text{opt}}[(\text{let}_p ((n (\text{block-get } b \ i))) e)]])$

$\mapsto_{\text{opt}}$   $(\text{let}_p ((b (\text{block-alloc-k})))$

$C_{\text{opt}}[(\text{let}_p ((t (\text{block-set! } b \ i))))$

$C'_{\text{opt}}[e\{n \rightarrow v\}]]])$

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

27

## Exercise

CPS/L<sub>3</sub> 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.

27

## CPS/L<sub>3</sub> inlining

28

## (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.

## Inlining

1. Shrinking:

```
(letf ((f1 e1) ... (fi (fun (ci ni,1 ...) ei)) ... (fk ek))  
  Copt[(appf fi c m1 ...)])  
→opt (letf ((f1 e1) ... (fk ek))  
      Copt[ei{ci→c}{ni,1→m1}...])  
[when fi is not free in Copt, e1, ..., en]
```

2. Non-shrinking (fresh versions of bound names should be created to preserve their global uniqueness):

```
(letf (... (fi (fun (ci ni,1 ...) ei)) ...)  
  Copt[(appf fi c m1 ...)])  
→opt (letf (... (fi (fun (ci ni,1 ...) ei)) ...)  
      Copt[ei{ci→c}{ni,1→m1}...])
```

## Inlining heuristics (1)

Heuristics must be used to decide when to perform non-shrinking 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<sub>3</sub> and why such an encoding would not suffer from the above problem.

## CPS/L<sub>3</sub> "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<sub>3</sub> 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<sub>3</sub> 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 `appf` 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:

```
(letf ((f (fun (c a1 ...) e)))
  Copt[C'opt[(appf f c0 n1,1 ...), (appf f c0 n2,1 ...), ...]]
  ↗opt Copt[(letc ((m (cnt (a1 ...) e{c→c0})))
    C'opt[(appc m n1,1 ...), (appc m n2,1 ...), ...]])]
```

where:

- `f` does not appear free in `Copt` or `C'opt`,
- `C'opt` is the smallest (multi-hole) context enclosing all applications of `f`,
- `c0` is the (single) return continuation that is passed to function `f`.

## Recursive contifiability

A set of mutually-recursive functions  $F = \{f_1, \dots, f_n\}$  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<sub>3</sub> translation of the following L<sub>3</sub> 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`  $\in F$ ,
- `even` is tail-called from `odd`  $\in F$  and called with the continuation of the `letrec` statement – the common return continuation `c0` for this example.

## Recursive contification

Given a set of mutually-recursive functions

$(\text{let}_f ((f_1 e_1) (f_2 e_2) \dots (f_n e_n)) e)$

the condition  $\text{Cnt}(F)$  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

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

$$\begin{aligned} & (\text{let}_f ((f_1 (\text{fun } (c_1 a_{1,1} \dots) e_1)) \dots (f_n \dots))) \\ & C_{\text{opt}}[e] \\ \rightsquigarrow_{\text{opt}} & (\text{let}_f ((f_{i+1} (\text{fun } (c_{i+1} a_{i+1,1} \dots) e_{i+1})) \dots (f_n \dots))) \\ & C_{\text{opt}}[(\text{let}_c ((m_1 (\text{cnt } (a_{1,1} \dots) \\ & \quad e_1^* \{c_1 \rightarrow c_0\})) \dots) \\ & \quad e^*))]) \end{aligned}$$

where  $f_1, \dots, f_i$  do not appear free in  $C_{\text{opt}}$  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  $F = \{f_1, \dots, f_i\}$  appear in the body of the function  $f_n$ , and  $\text{Cnt}(F)$  holds:

$$\begin{aligned} & (\text{let}_f ((f_1 (\text{fun } (c_1 a_{1,1} \dots) e_1)) \dots \\ & \quad (f_n (\text{fun } (c_n a_{n,1} \dots) C_{\text{opt}}[e_n])))) e) \\ \rightsquigarrow_{\text{opt}} & (\text{let}_f ((f_{i+1} (\text{fun } (c_{i+1} a_{i+1,1} \dots) e_{i+1})) \dots \\ & \quad (f_n (\text{fun } (c_n a_{n,1} \dots) \\ & \quad C_{\text{opt}}[(\text{let}_c ((m_1 (\text{cnt } (a_{1,1} \dots) \\ & \quad \quad e_1^* \{c_1 \rightarrow c_0\})) \dots) \\ & \quad \quad e_n^*])))) e) \end{aligned}$$

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

## Contifiable subsets

Given a  $\text{let}_f$  term defining a set of functions  $F = \{f_1, \dots, f_n\}$ , 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.  $\text{Cnt}(F_i)$ , or not contifiable at all – i.e. none of its subsets of functions are contifiable.