CS107: Optimization

Guannan Wei
guannan.wei@tufts.edu

November 3, 2026

Spring 2026

Tufts University

Recap

Last time:

- Optimization: rewrite program into an equivalent but more efficient form.
- There are machine-independent and -dependent optimizations
- We discussed the importance of intermediate representations (IRs) for implementing optimizations.

IR Comparison

Conclusions:

- a good IR should not allow multiple assignments to a variable,
- a good IR should only allow atomic arguments to functions.

Examine IRs we have discussed so far:

- 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 and MiniScala/CPS are good as variables are immutable, and they only allow atoms as function arguments.

Rewriting Optimizations

- Constant propagation
- Constant folding
- Dead-code elimination
- Common subexpression elimination
- Eta reduction
- Algebraic laws
- Function inlining
- Contification

Simple CPS Optimizations - Rewriting Optimizations

General setup of rewriting optimizations for MiniScala/CPS:

- Rewriting rules are specified in the form `T` $\mapsto_{opt}$ `U`. They rewrite a MiniScala/CPS term `T` to an equivalent and more efficient term `U`.
- Rewriting rules can only be applied in certain **optimization contexts**.
- Using the optimization rewriting rules (presented later) and the optimization contexts, we can specify the optimization relation opt that rewrites a term to an optimized version:

  $\texttt{C}_{opt}\texttt{[T]} \Rightarrow_{opt} \texttt{C}_{opt}\texttt{[U] where T} \mapsto_{opt} \texttt{U}$

Optimization Contexts

For example, it would be incorrect to try to rewrite the parameter list of a function. This constraint can be captured 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]` (same idea as we specified for CPS transformation).

Example

If C is `if` $\square$ `ct else cf`, then `C[x < y]` $=$ `if (x < y) ct else cf`.

Optimization Contexts

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

$$
\begin{aligned}
\texttt{C}_{opt} \texttt{ ::= } & \square \\
\mid\ & \texttt{val}_{l}\ \texttt{n = l; C}_{opt} \\
\mid\ & \texttt{val}_{p}\ \texttt{n = p(n}_{1}\texttt{, ...); C}_{opt} \\
\mid\ & \texttt{def}_{c}\ \texttt{c}_{1}\texttt{... ; def}_{c}\texttt{c}_{i}\texttt{(n}_{i,1}\texttt{, ...) = \{ C}_{opt}\texttt{ \}; ...; def}_{c}\texttt{c}_{k}\texttt{...; e} \\
\mid\ & \texttt{def}_{c}\ \texttt{c}_{1}\texttt{...; C}_{opt} \\
\mid\ & \texttt{def}_{f}\ \texttt{f}_{1}\texttt{... ; def}_{f}\texttt{f}_{i}\texttt{(c}_{i}\texttt{, n}_{i,1}\texttt{, ...) = \{ C}_{opt}\texttt{ \}; ...; def}_{c}\texttt{f}_{k}\texttt{...; e} \\
\mid\ & \texttt{def}_{f}\ \texttt{f}_{1}\texttt{...; C}_{opt}
\end{aligned}
$$

A $\texttt{C}_{opt}$ describes the surrounding of a term in which it is valid to perform an optimization rewrite.

(Non-)shrinking rules

We distinguish two classes of rewriting rules:

1) **Shrinking rules** rewrite a term to an equivalent but **smaller** one. Can be applied at will, possibly until the term is fully reduced.

(Non-)shrinking rules

We distinguish two classes of rewriting rules:

1) **Shrinking rules** rewrite a term to an equivalent but **smaller** one. Can be applied at will, possibly until the term is fully reduced.

2) **Non-shrinking rules** rewrite a term to an equivalent but **potentially larger** one. Cannot be applied at will, as they could result in infinite expansion. Heuristics must be used to decide when to apply non-shrinking rules.

(Non-)shrinking rules

We distinguish two classes of rewriting rules:

1) **Shrinking rules** rewrite a term to an equivalent but **smaller** one. Can be applied at will, possibly until the term is fully reduced.

2) **Non-shrinking rules** rewrite a term to an equivalent but **potentially larger** one. Cannot be applied at will, as they could result in infinite expansion. Heuristics must be used to decide when to apply non-shrinking rules.

Except for (one-time) 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.

```
val_l n1 = 42; // dead-code
val_l n2 = 10;
n2
```

Dead Code Elimination

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

Rules:

$$\texttt{val}_{l}\ \texttt{n = l; e} \mapsto_{opt} \texttt{e}\ [\ \texttt{if n is not free in e}\ ]$$

Dead Code Elimination

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

Rules:

$$\texttt{val}_l\ \texttt{n = l; e} \mapsto_{opt} \texttt{e [if n is not free in e]}$$

$$\texttt{val}_p\ \texttt{n = p(n}_1\texttt{, ...); e} \mapsto_{opt} \texttt{e [if n is not free in e and p} \notin \texttt{\{ byte-read, byte-write, block-set \}]}$$

Dead Code Elimination

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

Rules:

$$\texttt{val}_l\ \texttt{n = l; e} \mapsto_{opt} \texttt{e [if n is not free in e]}$$

$$\texttt{val}_p\ \texttt{n = p(}n_1\texttt{, ...); e} \mapsto_{opt} \texttt{e [if n is not free in e and p} \notin \texttt{\{ byte-read, byte-write, block-set \}]}$$

$$\texttt{def}_f\ \texttt{f}_1\texttt{...; def}_f\texttt{f}_i\texttt{...; def}_f\texttt{f}_k\texttt{...; e} \mapsto_{opt} \texttt{def}_f\texttt{f}_1\texttt{; ...; def}_f\texttt{f}_{i-1}\texttt{...; def}_f\texttt{f}_{i+1}\texttt{; def}_f\texttt{f}_k\texttt{...; e [if f}_i\texttt{ is not free in \{ f}_1\texttt{, ..., f}_{i-1}\texttt{, f}_{i+1}\texttt{, ..., f}_k\texttt{, e \}]}$$

The rule for continuations is similar to the one for functions

Dead Code Elimination

Can we remove the mutually-recursive functions `even` and `odd`?

```
def_f even(x: Int) = x == 0 || odd(x - 1);
def_f odd(x: Int) = x == 1 || even(x - 1);
0
```

Dead Code Elimination

Can we remove the mutually-recursive functions `even` and `odd`?

```
def_f even(x: Int) = x == 0 || odd(x - 1);
def_f odd(x: Int) = x == 1 || even(x - 1);
0
```

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.

Certainly

```
val_l n1 = 42 + 10;
val_l n2 = 42 + 10;
f(n1, n2)
```

can be optimized to

```
val_l n1 = 42 + 10;
f(n1, n1)
```

CSE

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

But also

```
val_l n1 = 42 + 10;
def_f g() = {
  val_l n2 = 42 + 10;
  f(n1, n2)
}
```

should be optimized too.

How should we express the optimization rule for this case?

CSE

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

$$\texttt{val}_l\ \texttt{n}_1\ \texttt{= l; C}_{opt}\texttt{[val}_l\texttt{n}_2\ \texttt{= l; e]}$$
$$\mapsto_{opt}\ \texttt{val}_l\texttt{n}_1\ \texttt{= l; C}_{opt}\texttt{[e[n}_2 \mapsto \texttt{n}_1\texttt{]]}$$

$$\texttt{val}_p\ \texttt{n}_1\ \texttt{= a}_1\ \texttt{+ a}_2\texttt{; C}_{opt}\texttt{[val}_p\texttt{n}_2\ \texttt{= a}_1\ \texttt{+ a}_2\texttt{; e]}$$
$$\mapsto_{opt}\ \texttt{val}_p\texttt{n}_1\ \texttt{= a}_1\ \texttt{+ a}_2\texttt{; C}_{opt}\texttt{[e[n}_2 \mapsto \texttt{n}_1\texttt{]]}$$

etc.

Recall that

$$\texttt{C}_{opt}\texttt{[T]} \Rightarrow_{opt}\ \texttt{C}_{opt}\texttt{[U] where T} \mapsto_{opt}\ \texttt{U}$$

so we allow optimizing cases such as $\texttt{C}^1_{opt}\texttt{[...}\ \ \texttt{C}^2_{opt}\texttt{[...]]}$.

CSE

These simple rules for common subexpression elimination are also a bit too naive and miss some opportunities because of scoping.

For example, the common subexpression `y + z` below is not optimized by them:

```
def_c c1() = {
  val_p n1 = y + z; ...
};
def_c c2() = {
  val_p n2 = y + z; ...
};
...
```

Eta Reduction

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

```
def g(x: Int) = ...
def f(x: Int) = g(x) // f is an η-expansion of g
f(42)
```

can be optimized to

```
def g(x: Int) = ...
g(42)
```

Eta Reduction

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

Formally:

```
def_c c_1(...) = { e_1 }; ...;
def_c c_i(n_1, ...) = { d(n_1, ...) }; // d is a continuation
def_c c_k(...) = { e_k }; e
↦_opt def_c c_1(...) = { e_1[c_i ↦ d] }; ...; def_c c_k(...) = { e_k[c_i ↦ d] };
        e[c_i ↦ d]

def_f f_1(...) = { e_1 }; ...;
def_f f_i(c, n_1, ...) = { g(c, n_1, ...) }; // g is a function
def_f f_k(...) = { e_k }; e
↦_opt def_f f_1(...) = { e_1[f_i ↦ g] }; ...; def_f f_k(...) = { e_k[f_i ↦ g] };
        e[f_i ↦ g]
```

Constant folding (1)

Constant folding replaces a constant expression by its value.

For example,

```
val_l n = 42;
val_l m = 10;
val_p k = n + m;
...
```

can be optimized to

```
val_l n = 42;
val_l m = 10;
val_l k = 52;
...
```

Constant folding (1)

Constant folding replaces a constant expression by its value.

Example for addition:

$\texttt{val}_l$ n$_1$ = l1;
  $\texttt{C}^1_{opt}$[$\texttt{val}_l$ n$_2$ = l2;
    $\texttt{C}^2_{opt}$[$\texttt{val}_p$ n3 = n$_1$ + n$_2$; e]]

$\mapsto_{opt}$

$\texttt{val}_l$ n$_1$ = l1;
  $\texttt{C}^1_{opt}$[$\texttt{val}_l$ n$_2$ = l2;
    $\texttt{C}^2_{opt}$[$\texttt{val}_l$ n3 = (l1 + l2); e]]

Similar rules exist for other arithmetic primitives.

Constant folding (2)

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

```
if (n == n) ct else cf ↦_opt ct()

if (n != n) ct else cf ↦_opt cf()

val_l n_1 = l1;
  C^1_opt[ val_l n_2 = l2;
    C^2_opt[ if (n_1 == n_2) ct else cf ]]
↦_opt
val_l n_1 = l1;
  C^1_opt[ val_l n_2 = l2;
    C^2_opt[ ct() ]] [ if l1 == l2 ]
```

Neutral/Absorbing Elements

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

$$\texttt{val}_l\ \texttt{n}_1\ \texttt{= 0; C}_{opt}\texttt{[val}_p\texttt{n = n}_1\ \texttt{* n}_2\texttt{; e]}$$
$$\mapsto_{opt}\ \texttt{val}_l\texttt{n}_1\ \texttt{= 0; C}_{opt}\texttt{[e[n} \mapsto \texttt{n}_1\texttt{]]}$$

$$\texttt{val}_l\ \texttt{n}_2\ \texttt{= 0; C}_{opt}\texttt{[val}_p\texttt{n = n}_1\ \texttt{* n}_2\texttt{; e]}$$
$$\mapsto_{opt}\ \texttt{val}_l\texttt{n}_2\ \texttt{= 0; C}_{opt}\texttt{[e[n} \mapsto \texttt{n}_2\texttt{]]}$$

$$\texttt{val}_l\ \texttt{n}_1\ \texttt{= 1; C}_{opt}\texttt{[val}_p\texttt{n = n}_1\ \texttt{* n}_2\texttt{; e]}$$
$$\mapsto_{opt}\ \texttt{val}_l\texttt{n}_1\ \texttt{= 1; C}_{opt}\texttt{[e[n} \mapsto \texttt{n}_2\texttt{]]}$$

$$\texttt{val}_l\ \texttt{n}_2\ \texttt{= 1; C}_{opt}\texttt{[val}_p\texttt{n = n}_1\ \texttt{* n}_2\texttt{; e]}$$
$$\mapsto_{opt}\ \texttt{val}_l\texttt{n}_2\ \texttt{= 1; C}_{opt}\texttt{[e[n} \mapsto \texttt{n}_1\texttt{]]}$$

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:

```
val_p b = block-alloc-k(s);
  C^1_opt[ val_p t = block-set(b, i, v);
    C^2_opt[ val_p n = block-get(b, i); e ]]
↦_opt
val_p b = block-alloc-k(s);
  C^1_opt[ val_p t = block-set(b, i, v);
    C^2_opt[ e[n ↦ v] ]]
[ when tag k identifies a block that is not modified after
initialization, e.g. a closure block ]
```

Exercise

MiniScala/CPS 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 some rewriting optimizations that could be performed on these primitives, and in which conditions they could be performed.

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:

$$\texttt{def}_f\ \texttt{f}_1\texttt{...;}\ \texttt{def}_f\texttt{f}_i\texttt{(c}_i\texttt{, n}_{i,1}\texttt{, ...) = \{ e}_i\texttt{ \};}\ \texttt{...;}\ \texttt{def}_f\texttt{f}_k\texttt{...;}$$
$$\texttt{C}_{opt}\texttt{[f}_i\texttt{(c, m}_1\texttt{, ...)]}$$

$$\mapsto_{opt}$$

$$\texttt{def}_f\ \texttt{f}_1\texttt{...;}\ \texttt{def}_f\texttt{f}_{i-1}\texttt{...;}\ \texttt{def}_f\texttt{f}_{i+1}\texttt{...;}\ \texttt{...;}\ \texttt{def}_f\texttt{f}_k\texttt{...;}$$
$$\texttt{C}_{opt}\texttt{[e}_i\texttt{[c}_i \mapsto \texttt{c, n}_{i,1} \mapsto \texttt{m}_1\texttt{]]}$$
$$\texttt{[when f}_i\texttt{ is not free in C}_{opt}\texttt{, e}_1\texttt{, ..., e}_k\texttt{]}$$

The rule for continuations is similar.

General Inlining

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

```
...; def_f f_i(c_i, n_{i,1}, ...) = { e_i }; ...;
C_opt[ f_i(c, m_1, ...) ]
```

$\mapsto_{opt}$

```
...; def_f f_i(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 general inlining.

Summary

- A number of simple optimizations, and how the choice of IR can make them easier or harder to implement.
- How to specify optimizations as rewrite rules, and the notion of optimization contexts.
- Project 6: implementing these optimizations is one of the options.
- Next time: dataflow analyses that enable optimizations in the presence of loops.