Code optimizations

Compilers: Principles And Practice

Tiark Rompf

Optimization

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

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

Optimizations can be broadly split into two classes:

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

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.

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.

Naive Inlining: Problem #1

```
def printReturn(x: Int) = { printInt(x); x };
def twice(y: Int) = y + y;
def f(z: Int) = twice(printReturn(z));
```

Incorrect inlining of `twice` in `f`:

```
def f(z: Int) = printReturn(z) + printReturn(z)
```

Naive Inlining: Problem #1

```
def printReturn(x: Int) = { printInt(x); x };
def twice(y: Int) = y + y;
def f(z: Int) = twice(printReturn(z));
```

Incorrect inlining of `twice` in `f`:

```
def f(z: Int) = printReturn(z) + printReturn(z)
```

z is printed twice!!!!

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

Naive Inlining: Problem #2

```
def first[T,U](t: T, u: U) = t;
def printReturn(z: Int) = first[Int,Unit](z, printInt(z));
```

Incorrect inlining of `first` in `printReturn`:

```
def printReturn(z: Int) = z
```

Naive Inlining: Problem #2

```
def first[T,U](t: T, u: U) = t;
def printReturn(z: Int) = first[Int,Unit](z, printInt(z));
```

Incorrect inlining of `first` in `printReturn`:

```
def printReturn(z: Int) = z
```

z is not 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/MiniScala and similar functional IRs are good in that their variables are immutable, and they only allow atoms as function arguments.

Simple CPS Optimizations - Rewriting Optimizations

The rewriting optimizations for CPS/MiniScala are specified as a set of rewriting rules of the form `T` $\mapsto_{opt}$ `U`.

These rules rewrite a CPS/MiniScala term `T` to an equivalent - but hopefully more efficient - term `U`.

Optimization Contexts

Optimization rewriting rules cannot be applied anywhere, but only in specific locations. 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]`.

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

Optimization Contexts

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

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

Optimization Relation

Using the optimization rewriting rules - presented later - and the optimization contexts, it is possible to specify the optimization relation opt that rewrites a term to an optimized version:

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

(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 could 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.

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

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

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

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.

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

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

etc.

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$_c$ c$_1$() = {
  val$_p$ n$_1$ = y + z; ...
};
def$_c$ c$_2$() = {
  val$_p$ n$_2$ = y + z; ...
};
...
```

Eta Reduction

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

```
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
$\mapsto_{opt}$ def$_c$ c$_1$(...) = { e$_1$[c$_i$ $\mapsto$ d] }; ...; def$_c$ c$_k$(...) = { e$_k$[c$_i$ $\mapsto$ d] };
        e[c$_i$ $\mapsto$ 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
$\mapsto_{opt}$ def$_f$ f$_1$(...) = { e$_1$[f$_i$ $\mapsto$ g] }; ...; def$_f$ f$_k$(...) = { e$_k$[f$_i$ $\mapsto$ g] };
        e[f$_i$ $\mapsto$ g]
```

Constant folding (1)

Constant folding replaces a constant expression by its value.

Example for addition:

$$
\begin{array}{l}
\mathbf{val}_l\ \mathtt{n}_1\ \mathtt{=}\ \mathtt{l1;}\\
\quad \mathtt{C}^1_{opt}[\ \mathbf{val}_l\ \mathtt{n}_2\ \mathtt{=}\ \mathtt{l2;}\\
\qquad \mathtt{C}^2_{opt}[\ \mathbf{val}_p\ \mathtt{n3}\ \mathtt{=}\ \mathtt{n}_1\ \mathtt{+}\ \mathtt{n}_2\mathtt{;}\ \mathtt{e}\ ]]\\
\mapsto_{opt}\\
\mathbf{val}_l\ \mathtt{n}_1\ \mathtt{=}\ \mathtt{l1;}\\
\quad \mathtt{C}^1_{opt}[\ \mathbf{val}_l\ \mathtt{n}_2\ \mathtt{=}\ \mathtt{l2;}\\
\qquad \mathtt{C}^2_{opt}[\ \mathbf{val}_l\ \mathtt{n3}\ \mathtt{=}\ \mathtt{(l1\ +\ l2);}\ \mathtt{e}\ ]]
\end{array}
$$

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 $\mapsto_{opt}$ ct()

if (n != n) ct else cf $\mapsto_{opt}$ 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]]
$\mapsto_{opt}$
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:

$$\begin{aligned}&\mathbf{val}_l\ n_1 = 0;\ C_{opt}[\ \mathbf{val}_p\ n = n_1 * n_2;\ e\ ]\\ &\mapsto_{opt}\ \mathbf{val}_l\ n_1 = 0;\ C_{opt}[\ e[n \mapsto n_1]\ ]\end{aligned}$$

$$\begin{aligned}&\mathbf{val}_l\ n_2 = 0;\ C_{opt}[\ \mathbf{val}_p\ n = n_1 * n_2;\ e\ ]\\ &\mapsto_{opt}\ \mathbf{val}_l\ n_2 = 0;\ C_{opt}[\ e[n \mapsto n_2]\ ]\end{aligned}$$

$$\begin{aligned}&\mathbf{val}_l\ n_1 = 1;\ C_{opt}[\ \mathbf{val}_p\ n = n_1 * n_2;\ e\ ]\\ &\mapsto_{opt}\ \mathbf{val}_l\ n_1 = 1;\ C_{opt}[\ e[n \mapsto n_2]\ ]\end{aligned}$$

$$\begin{aligned}&\mathbf{val}_l\ n_2 = 1;\ C_{opt}[\ \mathbf{val}_p\ n = n_1 * n_2;\ e\ ]\\ &\mapsto_{opt}\ \mathbf{val}_l\ n_2 = 1;\ C_{opt}[\ e[n \mapsto n_1]\ ]\end{aligned}$$

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:

```
$\mathbf{val}_p$ b = block-alloc-k(s);
  $C^1_{opt}$[ $\mathbf{val}_p$ t = block-set(b, i, v);
    $C^2_{opt}$[ $\mathbf{val}_p$ n = block-get(b, i); e ]]
$\mapsto_{opt}$
$\mathbf{val}_p$ b = block-alloc-k(s);
  $C^1_{opt}$[ $\mathbf{val}_p$ t = block-set(b, i, v);
    $C^2_{opt}$[ e[n $\mapsto$ v] ]]
[ when tag k identifies a block that is not modified after
initialization, e.g.  a closure block ]
```

Exercise

CPS/MiniScala 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.

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:

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

$\mapsto_{opt}$

$$\begin{array}{l}
\mathbf{def}_f\ \texttt{f}_1\texttt{...;}\ \mathbf{def}_f\ \texttt{f}_{i-1}\texttt{...;}\ \mathbf{def}_f\ \texttt{f}_{i+1}\texttt{...; ...;}\ \mathbf{def}_f\ \texttt{f}_k\texttt{...;} \\
\quad \texttt{C}_{opt}[\ \texttt{e}_i\texttt{[c}_i \mapsto \texttt{c, n}_{i,1} \mapsto \texttt{m}_1\texttt{]}\ ]
\end{array}$$

$[\ \texttt{when f}_i\texttt{ is not free in C}_{opt}\texttt{, e}_1\texttt{, ..., e}_k\ ]$

The rule for continuations is similar.

General Inlining

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

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

$$\mapsto_{opt}$$

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

To preserve the uniqueness of names, fresh versions of bound names should be created during general 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 (1)

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,

to be continued. . .

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/MiniScala and why such an encoding would not suffer from the above problem

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 MiniScala example below to a local continuation, leading to efficient compiled code.

```
def fact(x: Int) = {
  def loop(i: Int, res: Int): Int = {
    if (i > x)
      res
    else
      loop(i + 1, res * i)
  };
  loop(x, 1)
}
```

Contifiability

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

For a non-recursive function, this condition is satisfied if and only if that function is only used in `AppF` nodes, in function position, and always passed the same return continuation.

For recursive functions, the condition is slightly more involved - see later.

Non-Recursive Contification

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

$$\begin{array}{l}\mathbf{def}_f\ \mathtt{f(c,\ n_1,\ ...)\ =\ \{\ e\ \};}\\ \quad C^1_{opt}[\ C^2_{opt}[\ \mathtt{f(c_0,\ n_{1,1},\ ...),\ f(c_0,\ n_{2,1},\ ...),\ ...]]}\end{array}$$

$$\mapsto_{opt}$$

$$\begin{array}{l}C^1_{opt}[\ \mathbf{def}_c\ \mathtt{m(n_1,\ ...)\ =\ \{\ e[c} \mapsto \mathtt{c_0]\ \};}\\ \quad C^2_{opt}[\ \mathtt{m(n_{1,1},\ ...),\ m(n_{2,1},\ ...),\ ...]]}\end{array}$$

where f does not appear free in either C^1_{opt} or C^2_{opt}, and C^2_{opt} is the smallest (multi-hole) context enclosing all applications of f. It ensures that the scope of m is as small as possible, and therefore that m obeys the scoping rules for continuations.

In this rule, c_0 is the (single) return continuation that is passed to function `f`.

Recursive Contifiability

A set of mutually-recursive functions F $= \{$ f$_1$, . . . , fn $\}$ 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/MiniScala translation of the following MiniScala term are contifiable:

```
def_f even(x: Int) = x == 0 || odd(x - 1);
def_f odd(x: Int) = x != 0 || 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 c_0 for this example.

Recursive Contification

Given a set of mutually-recursive functions

```
def_f f_1...; ...; def_f f_k...;
e
```

the condition Cnt(F) for some $F \subseteq \{ f_1, \dots, f_k \}$ 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 $F = \{ f_1, \ldots, f_i \}$ appear in the body of the `LetRec`, and Cnt(F) holds, contification is performed by the following rewriting:

$\mathbf{def}_f$ $\mathtt{f_1(c_1,\ n_{1,1},\ \ldots) = \{\ e_1\ \};\ \ldots;}$ $\mathbf{def}_f$ $\mathtt{f_k\ldots;}$

$\quad \mathtt{C_{opt}[\ e\ ]}$

$\mapsto_{opt}$

$\mathbf{def}_f$ $\mathtt{f_{i+1}(c_{i+1},\ n_{i+1,1},\ \ldots) = \{\ e_{i+1}\ \};\ \ldots;}$ $\mathbf{def}_f$ $\mathtt{f_k\ldots;}$

$\quad \mathtt{C_{opt}[}$ $\mathbf{def}_f$$\mathtt{c\ m_1(n_{1,1},\ \ldots) = \{\ e_1^*[c_1 \mapsto c_0]\ \};\ e^*\ ]}$

[where $f_1, \ldots, f_i$ 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 $= \{ \text{f}_1, \ldots, \text{f}_i \}$ appear in the body of the function fn, and Cnt(F) holds, contification is performed by the following rewriting:

$\mathbf{def}_f\ \mathtt{f}_1(\mathtt{c}_1,\ \mathtt{n}_{1,1},\ \ldots) = \{\ \mathtt{e}_1\ \};\ \ldots;$
$\mathbf{def}_f\ \mathtt{f}_k(\mathtt{c}_k,\ \mathtt{n}_{k,1},\ \ldots) = \{\ \mathtt{C}_{opt}[\ \mathtt{e}_k\ ]\ \};\ \mathtt{e}$

$\mapsto_{opt}$

$\mathbf{def}_f\ \mathtt{f}_{i+1}(\mathtt{c}_{i+1},\ \mathtt{n}_{i+1,1},\ \ldots) = \{\ \mathtt{e}_{i+1}\ \};\ \ldots;$
$\mathbf{def}_f\ \mathtt{f}_k(\mathtt{c}_k,\ \mathtt{n}_{k,1},\ \ldots) = \{$
$\quad \mathtt{C}_{opt}[\ \mathbf{def}_c\ \mathtt{m}_1(\mathtt{n}_{1,1},\ \ldots) = \{\ \mathtt{e}^*_1[\mathtt{c}_1 \mapsto \mathtt{c}_0]\ \};\ \mathbf{def}_c\ \ldots;\ \mathtt{e}^*_k]$
$\};\ \mathtt{e}$

where $\text{f}_1, \ldots, \text{f}_i$ do not appear free in C_{opt} and e_k is minimal.

Contifiable Subsets

Given a `LetRec` term defining a set of functions F $= \{$ f$_1$, . . . , fn $\}$, 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.