Contification Using Dominators

Matthew Fluet
Cornell University
fluet@cs.cornell.edu

Stephen Weeks
InterTrust STAR Laboratories
sweeks@intertrust.com

ABSTRACT

Contification is a compiler optimization that turns a function that always returns to the same place into a continuation. Compilers for functional languages use contification to expose the control-flow information that is required by many optimizations, including traditional loop optimizations.

This paper gives a formal presentation of contification in MLton, a whole-program optimizing Standard ML compiler. We present two existing algorithms for contification in our framework, as well as a new algorithm based on the dominator tree of a program's call graph. We prove that the dominator algorithm is optimal. We present benchmark results on realistic SML programs demonstrating that contification has minimal overhead on compile time and significantly improves run time.

1. INTRODUCTION

Compiler writers for traditional imperative languages can choose from a vast array of well-understood optimizations to improve the quality of generated code. To avoid reinventing the wheel, compiler writers for functional languages should use these known techniques, or variants of them, whenever possible. In order to do this, they should use intermediate languages (ILs) that allow traditional optimizations to be applied with minimal changes. They should also ensure that their compiler translates source programs into the IL in a way that makes traditional optimizations applicable.

Traditional optimizations [1, 19] such as common-subexpression elimination, loop-invariant code motion, and global register allocation operate on a control-flow graph that represents intraprocedural information. Many traditional optimizations can be implemented efficiently using Static Single-Assignment form (SSA) [8], a convenient representation of def-use information in control-flow graphs. In contrast to traditional compilers, functional-language compilers typically use a λ-calculus based IL like Continuation-Passing Style (CPS) [2] or A-normal form [11]. Fortunately, the contrast is not as large as it would seem. It has been shown that functional programs and CPS are closely related to SSA [4, 14]. It is even possible to translate between a subset of CPS and SSA [14].

Functional languages use function calls to express all control-flow, including loops. In order to apply traditional optimizations, a functional language compiler must translate function calls in the source program into an IL that exposes the same intraprocedural control-flow information that is available in traditional SSA form. In particular, recursive functions used to implement loops in the source should be recognizable as loops in the IL. Section 8 of [14], which describes how to translate from CPS to SSA, contains the key to exposing intraprocedural control-flow information in functional languages.

> The idea is to find a set of procedures all of which are always called with the same continuation, and then to substitute that continuation for the procedures' continuation variables. The procedures are then themselves continuations.

What this means is that if a function always returns to the same place, then that function's calls and returns can be viewed as describing intraprocedural instead of interprocedural control-flow. As an example, consider the following functions:

```
fun g y = y - 1
fun f b = (if b then g 13 else g 15) + 1
```

and their translation into CPS:

```
fun g (y, k) = k (y - 1)
fun f (b, k) =
  let fun k' (x) = k (x + 1)
  in if b then g (13, k') else g (15, k')
  end
```

The declaration of `k'` defines a continuation that increments its argument and passes the result to the continuation of `f`. Since `g`'s continuation is always `k'`, we can transform `g` into a continuation within `f` and eliminate the continuation argument from the definition of `g` and from calls to `g`.

```
fun f (b, k) =
  let fun k' (x) = k (x + 1)
      fun g (y) = k' (y - 1)
  in if b then g (13) else g (15)
  end
```

This transformation exposes intraprocedural control flow information (`g` now directly calls `k'`), and enables subsequent optimization, such as inlining `k'`.

ICFP'01, September 3-5, 2001, Florence, Italy.

In this paper, we coin the term *contification* to mean turning functions into continuations. We also use the verb *contify*. Contification (under various names) has been used in functional-language compilers for over ten years [15] and has recently received attention in compilers for Dylan [20] and Moby [21]. Contification is also used in MLton [18], a compiler for Standard ML. A simple form of contification, tail-call optimization [19], has been used in traditional compilers for decades.

This paper gives a formal presentation of contification in MLton. First, we present an overview of the MLton compiler and of FOL, the first-order intermediate language on which contification is performed. Next, we formalize contification as the combination of an analysis and a transformation. Our framework allows a variety of analyses, and defines a single transformation that works for any of them. An analysis determines which functions in the program can be viewed as a continuation and the continuations to which they always return. The transformation uses the results of the analysis to rewrite the program, turning functions into continuations. The main contribution of the paper is the presentation of analyses derived from the algorithms used in a previous version of MLton and in the Moby compiler, as well as a new contification analysis that uses the dominator tree of a program's call graph. We prove that the dominator analysis is optimal, in the sense that it contifies any function that is contifiable by any other approach expressible in our framework. Finally, we describe an implementation of all of these algorithms as part of MLton, and provide measurements of their running time and effectiveness.

2. MLTON

This section presents an overview of MLton and describes where contification fits in the compilation process. MLton is a whole-program optimizing compiler for SML where the main focus has been the generation of efficient code. It does not support separate compilation. MLton is freely available under the GPL from `http://www.sourcelight.com/MLton`.

There is a large gap between SML and traditional ILs that use control flow because of SML features like parametric modules, polymorphism, and first-class functions. MLton relies on having the whole program to translate these features into a simply-typed first-order intermediate language, FOL, that is similar to SSA. MLton performs most optimizations on FOL, including contification.

We now describe the relevant compiler passes of MLton. First, MLton eliminates module level constructs from the input SML program. It removes all uses of structures and signatures by moving declarations to the top level and appropriately renaming variables. It removes functors by applying them at compile time, duplicating their bodies for each use [10]. This produces a program in a polymorphic, higher-order IL (the XML of [12]). Next, MLton eliminates polymorphism by duplicating each polymorphic expression for each monotype at which it is used. This produces a program in a simply typed, higher-order IL.

MLton then performs a monovariant whole-program flow analysis [13] to determine where each function could be called. MLton uses the results of the flow analysis to eliminate higher-order functions by the closure conversion algorithm described in [6]. First-class functions become data structures and calls to parameters of functional type become ordinary function calls, possibly preceded by a case dispatch.

$$
\begin{array}{lcl}
c & \in & \mathit{Const} \\
p & \in & \mathit{Prim} \\
k & \in & \mathit{Cont} \\
x & \in & \mathit{Var} \\
f & \in & \mathit{Func} \\
\\
P & ::= & \texttt{let fun } f\texttt{(}\vec{x}\texttt{) = } e \texttt{ ... in } f_{\mathsf{m}}\texttt{() end} \\
e & ::= & \texttt{let val } x \texttt{ = } s \texttt{ in } e \texttt{ end} \\
 & | & \texttt{let cont } k\texttt{(}\vec{x}\texttt{) = } e \texttt{ ... in } e \texttt{ end} \\
 & | & k\texttt{(}\vec{x}\texttt{)} \\
 & | & \texttt{if } x \texttt{ then } k_1\texttt{() else } k_2\texttt{()} \\
 & | & f\texttt{(}\vec{x}\texttt{)} \\
 & | & k\texttt{(}f\texttt{(}\vec{x}\texttt{))} \\
 & | & \vec{x} \\
s & ::= & c \\
 & | & x \\
 & | & \texttt{(}\vec{x}\texttt{)} \\
 & | & \texttt{\#}i\ x \\
 & | & p\texttt{(}\vec{x}\texttt{)}
\end{array}
$$

Figure 1: FOL syntax

The resulting program is in FOL, and is simply typed and first order. Next, MLton performs optimizations including contification.

All of MLton's ILs up to and including FOL are typed ILs [22]. All of the transformations and optimizations to this point, including contification, take well-typed programs as input and produce well-typed programs as output. In debugging mode, MLton runs a type checker on each IL after each optimization. After FOL optimization, MLton translates to a low level untyped IL, and then into C or native x86 instructions.

MLton is similar in structure to Tolmach and Oliva's RML-to-3GL translator [24]. However, this structure differs from that of most other functional language compilers such as Orbit [16], SML/NJ [2], and TIL [22] in three crucial ways. First, MLton is a whole-program compiler. This enables certain optimizations like monomorphisation and ensures that all optimization passes have access to more information. Second, MLton performs closure conversion early in the compilation process, before most optimization occurs. As a consequence, optimizations (including contification) operate on a very simple IL that is closer to more traditional ILs. Third, MLton uses whole-program flow analysis to drive closure conversion, and the results are expressed directly in FOL. Thus, optimizations (including contification) benefit from the control-flow information computed by the analysis. Optimizations need not recompute control flow information as is done in many analyses (e.g. [25]). Nor do they need to introduce imprecisions based on "escaping" functions, as is done in [21].

3. FOL

This section describes a slightly simplified version of MLton's FOL (exception handling constructs are omitted) and gives some examples of contification. FOL is very similar to Tolmach and Oliva's "First-order SIL with jump points" [24], which they use as the target language for tail recursion elimination. Figure 1 presents the syntax of FOL, which is lexically scoped, first order, and simply typed. We do not present the typing rules. We assume mutually disjoint sets of constants *Const*, primitives *Prim*, continuation labels *Cont*, variables *Var*, and function names *Func*. A program P declares a collection of mutually recursive functions and calls a distinguished nullary main function f_{m}. Functions can take multiple arguments and return multiple results. We use $\vec{x}$ to indicate a sequence of variables. We assume all variable, function, and continuation names are unique.

An expression e either binds a simple value, declares a continuation, or transfers control to another expression. Simple values are constants, variables, tuples, selections from tuples, or applications of primitives. The syntax `cont` $k(\vec{x})$ `=` e declares continuation k with arguments $\vec{x}$ and body e. Continuations can be declared simultaneously, in which case they are mutually recursive. Every expression finishes by transferring control, either via a jump to a local continuation $k(\vec{x})$, an if-then-else, a tail call $f(\vec{x})$, a nontail call $k(f(\vec{x}))$, or a return to its caller.

To keep our examples easy to read, we assume the existence of needed constants and primitives, write primitive applications in infix notation, and elide `val` bindings for constants and primitive applications. The example from Section 1 would be written in FOL as follows.

```
fun g (y) = y - 1
fun f (b) =
  let cont k' (x) = x + 1
      cont l1 () = k' (g (13))
      cont l2 () = k' (g (15))
  in if b then l1 () else l2 ()
  end
```

FOL is similar to SSA in the manner described in [4]. An FOL continuation declaration `cont` $k(\vec{x})$ `=` e is like a basic block with label k that executes the sequence of `val` bindings in e and transfers control according to the last expression in e. There two differences between SSA and FOL are that FOL uses lexical scoping to enforce the SSA condition that a variable definition dominate all of its uses and that FOL uses continuation call $k(\vec{x})$ to express the ϕ function that assigns the actuals $\vec{x}$ to the formals of k.

FOL is also similar to CPS, although this may not be apparent at first sight. After all, FOL has returns and nontail calls, and functions are not passed continuations as arguments. However, these are only minor syntactic differences, as was observed in [11]. The above example might be written in more traditional CPS as follows.

```
fun g (y, k) = k (y - 1)
fun f (b, k) =
  let cont k' (x) = k (x + 1)
      cont l1 () = g (13, k')
      cont l2 () = g (15, k')
  in if b then l1 () else l2 ()
  end
```

In FOL, the only use of continuations passed as arguments to functions would be to return as in `k (x + 1)` or in tail calls, which are not shown above. Also, FOL nontail calls such as `k' (g 13)` exactly correspond to traditional CPS tail calls with a nontrivial continuation `g (13, k')`, thus explaining our choice of the `cont` keyword. The FOL syntax simply elides redundant places where a continuation variable might be written (formal parameter, return, tail call) and writes nontail calls in a different order.

Continuing with the example in FOL, the result of contifying `g` within `f` would look like the following.

```
fun f (b) =
  let cont k' (x) = x + 1
      cont g (y) = k' (y - 1)
      cont l1 () = g (13)
      cont l2 () = g (15)
  in if b then l1 () else l2 ()
  end
```

Contification has transformed nontail calls to the function `g` into jumps to the continuation `g`. As a final example, here is an SML function to sum the elements in a vector.

```
fun sum (v) =
  let fun loop (i, s) =
        if i = length (v)
        then s
        else loop (i + 1, s + sub (v, i))
  in loop (0, 0)
  end
```

Here are the FOL functions that MLton would produce.

```
fun sum (v) = loop (v, 0, 0)
fun loop (v', i, s) =
  let cont l1 () = s
      cont l2 () = let val x = sub (v', i)
                   in loop (v', i + 1, s + x)
                   end
      val n = length (v')
  in if i = n then l1 else l2
  end
```

Contifying `loop` within `sum` yields the following, which demonstrates how recursive continuations express loops.

```
fun sum (v) =
  let cont loop (v', i, s) =
       let cont l1 () = s
           cont l2 () = let val x = sub (v', i)
                        in loop (v', i + 1, s + x)
                        end
           val n = length (v')
       in if i = n then l1 else l2
       end
   in loop (v, 0, 0)
   end
```

Although contification may appear to be like inlining, it is different. Inlining a function replaces a call to a function by its body, substituting actual arguments for formal parameters. On the other hand, contification does not duplicate code – it only moves code from one place to another, exposing control-flow information. Contification of the above example exposes the information that both the inner and outer call to `loop` always return to `sum`.

From the above example, contification might appear to be a “clean-up” optimization that is undoing the effects of MLton’s previous conversion of the program into FOL. After all, contification has just moved `loop` back inside `sum` after closure conversion took it out. Nevertheless, once we explain the semantics of FOL in the next section, we can show that contification is expressing a nontrivial fact about the program needed in order to perform some optimizations, and is independent of closure conversion.

$$
\begin{array}{rcrcl}
v & \in & \mathit{Value} & = & \mathit{Const} + \mathit{Value}^* \\
\rho & \in & \mathit{Env} & = & \mathit{Var} \rightarrow \mathit{Value} \\
\kappa & \in & \mathit{Stack} & = & (\mathit{Cont} \times \mathit{Env})^* \\
s & \in & \mathit{State} & = & \mathit{Exp} \times \mathit{Env} \times \mathit{Stack} \\
\\
 & & \longrightarrow & \subseteq & \mathit{State} \times \mathit{State}
\end{array}
$$

$$
\begin{array}{llll}
\textbf{(const)} & \langle \texttt{let val } x \texttt{ = } c \texttt{ in } e \texttt{ end}, \rho, \kappa \rangle & \longrightarrow \langle e, \rho[x \rightarrow c], \kappa \rangle & \\
\textbf{(var)} & \langle \texttt{let val } x \texttt{ = } y \texttt{ in } e \texttt{ end}, \rho, \kappa \rangle & \longrightarrow \langle e, \rho[x \rightarrow \rho(y)], \kappa \rangle & \\
\textbf{(tuple)} & \langle \texttt{let val } x \texttt{ = (} \vec{y} \texttt{) in } e \texttt{ end}, \rho, \kappa \rangle & \longrightarrow \langle e, \rho[x \rightarrow \rho(\vec{y})], \kappa \rangle & \\
\textbf{(select)} & \langle \texttt{let val } x \texttt{ = \#} i\ y \texttt{ in } e \texttt{ end}, \rho, \kappa \rangle & \longrightarrow \langle e, \rho[x \rightarrow \pi_i(\rho(y))], \kappa \rangle & \\
\textbf{(primapp)} & \langle \texttt{let val } x \texttt{ = } p\texttt{(}\vec{x}\texttt{) in } e \texttt{ end}, \rho, \kappa \rangle & \longrightarrow \langle e, \rho[x \rightarrow \delta(p, \rho(\vec{x}))], \kappa \rangle & \\
\textbf{(cont)} & \langle \texttt{let cont } k\texttt{(}\vec{x}\texttt{) = } e \ \ldots \texttt{ in } e' \texttt{ end}, \rho, \kappa \rangle & \longrightarrow \langle e', \rho, \kappa \rangle & \\
\textbf{(jump)} & \langle k\texttt{(}\vec{x}\texttt{)}, \rho, \kappa \rangle & \longrightarrow \langle e, \rho[\vec{y} \rightarrow \rho(\vec{x})], \kappa \rangle & \texttt{cont } k\texttt{(}\vec{y}\texttt{) = } e \in P \\
\textbf{(if-true)} & \langle \texttt{if } x \texttt{ then } k_1\texttt{() else } k_2\texttt{()}, \rho, \kappa \rangle & \longrightarrow \langle e, \rho, \kappa \rangle & \rho(x) = \mathsf{true} \text{ and } \texttt{cont } k_1\texttt{() = } e \in P \\
\textbf{(if-false)} & \langle \texttt{if } x \texttt{ then } k_1 \texttt{ else } k_2, \rho, \kappa \rangle & \longrightarrow \langle e, \rho, \kappa \rangle & \rho(x) = \mathsf{false} \text{ and } \texttt{cont } k_2\texttt{() = } e \in P \\
\textbf{(tail)} & \langle f\texttt{(}\vec{x}\texttt{)}, \rho, \kappa \rangle & \longrightarrow \langle e, [\vec{y} \rightarrow \rho(\vec{x})], \kappa \rangle & \texttt{fun } f\texttt{(}\vec{y}\texttt{) = } e \in P \\
\textbf{(nontail)} & \langle k\texttt{(}f\texttt{(}\vec{x}\texttt{))}, \rho, \kappa \rangle & \longrightarrow \langle e, [\vec{y} \rightarrow \rho(\vec{x})], \langle k, \rho \rangle :: \kappa \rangle & \texttt{fun } f\texttt{(}\vec{y}\texttt{) = } e \in P \\
\textbf{(return)} & \langle \vec{x}, \rho, \langle k, \rho' \rangle :: \kappa \rangle & \longrightarrow \langle e, \rho'[\vec{y} \rightarrow \rho(\vec{x})], \kappa \rangle & \texttt{cont } k\texttt{(}\vec{y}\texttt{) = } e \in P
\end{array}
$$

Figure 2: FOL operational semantics

In the remainder of this paper, we write program fragments in which we show declarations and calls of relevant functions, but elide continuation declarations. We will always write the full expression for a control transfer, distinguishing between tail and nontail calls. For example, in the following program fragment, `fm` includes a nontail call to `f` and a tail call to `g`.

```
fun fm () = ... k (f ()) ... g () ...
fun f () = ... f () ...
fun g () = ...
```

3.1 Operational semantics

Figure 2 presents an operational semantics for programs in FOL via a transition relation $\longrightarrow$, written in infix. Technically speaking, $\longrightarrow$ is dependent on the program and should be written $\longrightarrow_P$, but since the program is always clear from context, we will drop the P. Values v are either constants or tuples of other values. An environment ρ is a map from variables to values. We write $\rho[x \rightarrow v]$ to denote the environment ρ' such that $\rho'(x) = v$ and $\rho'(y) = \rho(y)$ if $y \neq x$. We extend this notation to sequences of variables and values as in $\rho(\vec{x})$ and $\rho[\vec{x} \rightarrow \vec{v}]$, in which the sequences $\vec{x}$ and $\vec{v}$ must be of the same length. A state $\langle e, \rho, \kappa \rangle$ corresponds to the evaluation of expression e in environment ρ with a stack of callers κ to which e should return when done. A stack is a sequence of frames written with an infix cons operator as $\langle k, \rho \rangle :: \kappa$, where $\langle k, \rho \rangle$ is the top frame on the stack and κ is the rest of the stack. We write $\texttt{fun } f\texttt{(}\vec{x}\texttt{) = } e \in P$ or $\texttt{cont } k\texttt{(}\vec{x}\texttt{) = } e \in P$ to mean that the function or continuation declaration is in the program P.

The rules for simple expressions (const, var, tuple, select, and primapp) are all straightforward – each adds a binding to the environment and continues with the rest of the expression. The primapp rule assumes the existence of a function $\delta : \mathit{Prim} \times \mathit{Value}^* \rightarrow \mathit{Value}$, which defines the meaning of primitives. The rule for a continuation declaration does nothing, since continuations are just labels. The rules for local control flow transfers (jump, if-true, if-false) switch to the body of the desired continuation, possibly modifying the current environment. The rules rely on on the fact that all variable names are distinct in order to avoid capture of free variables in continuations. The rules for function call (tail and nontail) switch to the body of the desired function in a new environment, binding only the formal parameters of the function. The nontail rule adds a frame to the stack; the tail rule does not. The return rule pops the top frame off the stack, and continues with the expression corresponding to the continuation on the top frame.

The transition rules bring out an important difference between continuations and functions. Jumps to continuations maintain in the current environment while function calls (tail and nontail) create a new environment. Also, unlike traditional CPS, no closure needs to be created for a continuation, since its free variables can be found in the current environment (section 5.3 of [15]). The reason that continuations do not need their own environment is that the syntax of FOL guarantees that the environment at their point of definition always coincides with the environment at a point of use. Therefore, one can represent the environment as a stack frame and think of FOL variables as stack slots and continuations as basic blocks. With this in mind, we can see that the effect of contification is to turn function calls into jumps, thus exposing intraprocedural control-flow information. Also, when a function g is contified within a function f, it will be able to share the same stack frame with f. Finally, invocations of the contified function can be optimized as intraprocedural control-flow transfers, where live variables can be passed in registers, rather than as interprocedural control-flow transfers, where standard calling conventions must be followed.

With the semantics in hand, we can now see that in the `loop` example of the previous section, contification expresses the fact that `loop` always returns to `sum`. Hence, it can be implemented as a loop that shares the same stack frame as `sum`. The fact that `loop` always returns to `sum` is apparent in the source program, because `loop` always calls itself in tail position and there is only one outer call. This information is neither obscured nor clarified by MLton's translation of the source program into FOL. In fact, the contification analysis in Section 5.1 uses exactly that reasoning to contify `loop`.

It is possible to perform contification analysis on the source program, before closure conversion. However, in doing so one would encounter two problems: what to do about higher-order functions and how to express the results of the analysis. The first problem could be solved by introducing imprecision into the analysis whenever higher-order functions obscure control flow or by doing control-flow analysis along with contification. In MLton, we take the approach of performing control-flow analysis before contification, thus making its results available to contification. The second problem could be solved by annotating the program with the results of contification analysis in some way. This then leads to the question of how subsequent optimizations would use the annotation. We do not take this approach in MLton. As a practical matter in compiler construction, we find it easier to use an analysis if its results are expressed not as an annotation on the source program but instead via a program transformation into an IL whose semantics directly reflect the results of the analysis. Thus, in MLton and in this paper, we use contification analysis to transform the program into FOL, which has a very different semantics for function call versus continuation call.

4. CONTIFICATION

In this section, we present the two components of contification: analysis and transformation. A contification analysis specifies what is contified, and where. The transformation rewrites the program based on the analysis. To simplify the presentation, we abstract from FOL and represent a program by its call graph. A program $P = (f_{\mathsf{m}}, \mathrm{N}, \mathrm{T})$ consists of a main function, a multiset of nontail calls, and a multiset of tail calls. A nontail call consists of a caller, callee, and a continuation. A tail call consists of a caller and a callee. Formally, we have the following, where we write $\mathcal{M}(S)$ for the set of all multisets of S.

$$\begin{array}{rcrcl} P & \in & \mathit{Program} & = & \mathit{Func} \times \mathcal{M}(\mathit{Nontail}) \times \mathcal{M}(\mathit{Tail}) \\ & & \mathit{Nontail} & = & \mathit{Func} \times \mathit{Func} \times \mathit{Cont} \\ & & \mathit{Tail} & = & \mathit{Func} \times \mathit{Func} \end{array}$$

For each $k \in \mathit{Cont}$, there is a unique $f \in \mathit{Func}$ such that k is declared in the body of f; we write this f as $\mathcal{D}(k)$. We define the predicate $\mathcal{R} : \mathit{Func} \to \mathit{Bool}$ such that $\mathcal{R}(f)$ if and only if there is a path of calls from f_{m} to f in P.

4.1 Analysis

An analysis is a map from functions to abstract return points.

$$\begin{array}{rcrcl} \rho & \in & \mathit{Return} & = & \{\mathsf{Uncalled}, \mathsf{Unknown}\} \cup \mathit{Cont} \cup \mathit{Func} \\ \mathcal{A} & \in & \mathit{Analysis} & = & \mathit{Func} \to \mathit{Return} \end{array}$$

The following table describes the intended meaning of $\mathcal{A}(f)$, for $f \in \mathit{Func}$.

$\mathcal{A}(f)$	Meaning
Uncalled	f is never called during evaluation of P
$k \in \mathit{Cont}$	f always returns to continuation k
$g \in \mathit{Func}$	f always returns to function g
Unknown	f returns to multiple k's and/or g's

The meaning of $\mathcal{A}(f) = k$ is that whenever the body of f is evaluating, the top frame on the stack will have k as its continuation. The meaning of $\mathcal{A}(f) = g$ is that g is always responsible for calling f, either directly or through an intervening sequence of tail calls. We do not allow analyses to express information about sets of continuations other than to use Unknown, which represents the set of all continuations.

The contification transformation uses an analysis $\mathcal{A}$ as follows. The transformed program has as its functions those f with $\mathcal{A}(f) = \mathsf{Unknown}$. Functions with $\mathcal{A}(f) = \mathsf{Uncalled}$ are removed from the program. Functions with $\mathcal{A}(f) \in \mathit{Cont} \cup \mathit{Func}$ are contified in other functions. For $g \in \mathit{Func}$, the functions f such that $\mathcal{A}(f) = g$ are contified as the first declaration in the body of g. For $k \in \mathit{Cont}$, the functions f such that $\mathcal{A}(f) = k$ are contified in $\mathcal{D}(k)$ as the first declaration after the declaration of k.

We now introduce a condition on an analysis that ensures it will lead to a sensible transformation.

Definition 1. An analysis $\mathcal{A}$ is *safe* for a program $P = (f_{\mathsf{m}}, \mathrm{N}, \mathrm{T})$ if all of the following hold.

$*_1$ if $\neg\mathcal{R}(f)$, then $\mathcal{A}(f) = \mathsf{Uncalled}$.

$*_2$ $\mathcal{A}(f_{\mathsf{m}}) = \mathsf{Unknown}$.

$*_3$ for all nontail calls $(f, g, k) \in \mathrm{N}$,
if $\mathcal{R}(f)$ then $\mathcal{A}(g) \in \{k, \mathsf{Unknown}\}$.

$*_4$ for all tail calls $(f, g) \in \mathrm{T}$ with $f \neq g$,
if $\mathcal{R}(f)$ then $\mathcal{A}(g) \in \{f, \mathcal{A}(f), \mathsf{Unknown}\}$.

Another way to think of safety is that it ensures that the analysis conservatively approximates the actual run time behavior of the program, in the sense of abstract interpretation [7]. Condition $*_1$ forces unreachable functions to be marked Uncalled, which causes the transformation to remove them from the program. We need $*_1$ to ensure that the transformation is well-defined. Condition $*_2$ forces the main function to be marked Unknown, which prevents it from being contified. Condition $*_3$ forces the callee of a nontail call to return to the continuation of the call or be Unknown. Condition $*_4$ forces the callee of a tail call to return to the caller, have the same continuation as the caller, or be Unknown.

In the remainder of this paper, we will drop the phrase "for a program P" from definitions and theorems. Any two analyses mentioned in the same context will be considered analyses for the same program. The following lemma shows that a safe analysis never labels a reachable function Uncalled.

LEMMA 1. *If $\mathcal{A}$ is a safe analysis, then $\neg\mathcal{R}(f)$ iff $\mathcal{A}(f) =$ Uncalled.*

Although we can transform a program based on any safe analysis, there are many safe analyses with a wide range of utility. For example, the following trivial analysis is safe.

$$\mathcal{A}_{\mathrm{Triv}}(f) = \begin{cases} \mathsf{Uncalled} & \text{if } \neg\mathcal{R}(f) \\ \mathsf{Unknown} & \text{if } \mathcal{R}(f) \end{cases}$$

The transformation based on $\mathcal{A}_{\mathrm{Triv}}$ fails to do anything other than eliminate dead code. In order for the transformation to be useful (i.e., actually contify functions), it must be based on an analysis $\mathcal{A}$ with $\mathcal{A}(f) \neq \mathsf{Unknown}$. Hence, we are motivated to search for an analysis where $\mathcal{A}(f) \neq \mathsf{Unknown}$ for as many functions as possible. This leads us to introduce the definition of a *maximal* analysis.

$$
\begin{array}{l}
\mathcal{P} : \mathit{Program} \rightarrow \mathit{Program} \\
\mathcal{P}(\texttt{let } \{\texttt{fun } f_i(\vec{x}_i) \texttt{ = } e_i \mid i \in I\} \texttt{ in } f_{\mathsf{m}}\texttt{() end}) = \\
\quad \texttt{let } \{\texttt{fun } f_i(\vec{x}_i) \texttt{ = } \mathcal{F}(f_i, \bot) \mid i \in I \text{ and } \mathcal{A}(f_i) = \mathsf{Unknown}\} \texttt{ in } f_{\mathsf{m}}\texttt{() end}
\end{array}
$$

$$
\begin{array}{l}
\mathcal{F} : \mathit{Func} \times \mathit{Cont}_\bot \rightarrow \mathit{Exp} \\
\mathcal{F}(g, \underline{k}) = \texttt{let } \{\texttt{cont } \mathcal{K}(f)(\vec{x}) \texttt{ = } \mathcal{F}(f, \underline{k}) \mid \mathcal{A}(f) = g \text{ and } \texttt{fun } f(\vec{x}) \texttt{ = } e' \in P\} \texttt{ in } \mathcal{E}(e, \underline{k}) \texttt{ end} \\
\qquad\qquad \text{where } \texttt{fun } g(\vec{x}) \texttt{ = } e \in P
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathcal{E} : \mathit{Exp} \times \mathit{Cont}_\bot \rightarrow \mathit{Exp}} \\
\mathcal{E}(\texttt{let val } x \texttt{ = } s \texttt{ in } e \texttt{ end}, \underline{k}) & = & \texttt{let val } x \texttt{ = } s \texttt{ in } \mathcal{E}(e, \underline{k}) \texttt{ end} \\
\mathcal{E}(\texttt{let } \{\texttt{cont } k_i(\vec{x}_i) \texttt{ = } e_i \mid i \in I\} \texttt{ in } e \texttt{ end}, \underline{k}) & = & \texttt{let } \{\texttt{cont } k_i(\vec{x}_i) \texttt{ = } \mathcal{E}(e_i, \underline{k}) \mid i \in I\} \cup \\
 & & \qquad \bigcup_{i \in I} \{\texttt{cont } \mathcal{K}(f_{ij})(\vec{x}_{ij}) \texttt{ = } \mathcal{F}(f_{ij}, k_i) \mid j \in J_i\} \\
 & & \texttt{in } \mathcal{E}(e, \underline{k}) \\
 & & \texttt{end} \\
 & & \quad \text{where for all } i \in I,\ \{f_{ij} \mid j \in J_i\} = \{f \mid \mathcal{A}(f) = k_i\} \\
 & & \quad \text{and for all } j \in J_i,\ \texttt{fun } f_{ij}(\vec{x}_{ij}) \texttt{ = } e_{ij} \in P \\
\mathcal{E}(k(\vec{x}), \underline{k}) & = & k(\vec{x}) \\
\mathcal{E}(\texttt{if } x \texttt{ then } k_1\texttt{() else } k_2\texttt{()}, \underline{k}) & = & \texttt{if } x \texttt{ then } k_1\texttt{() else } k_2\texttt{()} \\
\mathcal{E}(f(\vec{x}), \underline{k}) & = & \text{if } \mathcal{A}(f) \neq \mathsf{Unknown} \text{ then } \mathcal{K}(f)(\vec{x}) \text{ else if } \underline{k} = k \text{ then } k(f(\vec{x})) \text{ else } f(\vec{x}) \\
\mathcal{E}(k(f(\vec{x})), \underline{k}) & = & \text{if } \mathcal{A}(f) \neq \mathsf{Unknown} \text{ then } \mathcal{K}(f)(\vec{x}) \text{ else } k(f(\vec{x})) \\
\mathcal{E}(\vec{x}, \underline{k}) & = & \text{if } \underline{k} = k \text{ then } k(\vec{x}) \text{ else } \vec{x}
\end{array}
$$

Figure 3: The contification transformation

Definition 2. A safe analysis $\mathcal{A}$ is *maximal* if for all safe analyses $\mathcal{B}$ and all functions $f \in \mathit{Func}$, $\mathcal{B}(f) \neq \mathsf{Unknown} \Rightarrow \mathcal{A}(f) \neq \mathsf{Unknown}$.

In Section 5.3, we will prove by construction that for all programs there exists a maximal safe analysis. A maximal analysis need not be unique – the following program fragment shows why.

```
fun fm () = ... k (f ()) ...
fun f () = ... g () ...
fun g () = ...
```

For this program, the analyses $\mathcal{A}_1$ and $\mathcal{A}_2$ below are both safe and maximal.

	$\mathcal{A}_1$	$\mathcal{A}_2$
fm	Unknown	Unknown
f	k	k
g	f	k

4.2 Transformation

Figure 3 presents the formal definition of the contification transformation. $\mathcal{P}(P)$ removes uncalled and contified functions and processes the remaining ones. We abuse FOL syntax slightly and use set notation to indicate collections of simultaneous function (or continuation) declarations. $\mathcal{F}(g, \underline{k})$ prefixes the set of functions that are contified in g onto the transform of g's body. We define $\mathit{Cont}_\bot = \mathit{Cont} \cup \{\bot\}$ and use $\underline{k}$ to range over $\mathit{Cont}_\bot$. For a function f that is contified, we need a fresh element of Cont to replace it; we write this element as $\mathcal{K}(f)$.

$\mathcal{E}(e, \underline{k})$ transforms expression e, changing calls and returns according to context $\underline{k}$. If $\underline{k} = k$, then the expression is transformed so that it transfers control to the continuation k. If $\underline{k} = \bot$, then control can be transferred as before. The rule for transforming a set of continuation declarations translates the bodies of the continuations and inserts all functions that are contified at one of the continuations.

The rule for a tail call depends on the context and whether or not the callee is contified. If the callee is contified ($\mathcal{A}(f) \neq \mathsf{Unknown}$), then the tail call is replaced by a jump to the contified function. In this case, the context is irrelevant because the body of the callee was transformed with the proper context. On the other hand, if the callee is not contified, then the transformation depends on the context $\underline{k}$. If $\underline{k} = k$, then there is a continuation to which f must return, so the tail call becomes a nontail call with continuation k. On the other hand, if $\underline{k} = \bot$, then there is no continuation to which f must return, so the tail call remains a tail call. The rule for a nontail call is based on $*_3$, which ensures that if f is a nontail callee and $\mathcal{A}(f) \neq \mathsf{Unknown}$, then $\mathcal{A}(f) = k$. The rule for a return replaces the return by a jump when the context requires it.

4.3 Related Transformations

Contification is superficially similar to lambda dropping [9], which also nests functions. However, they are complementary optimizations. Roughly speaking, the "block sinking" component of lambda dropping uses the call graph to nest a function f in another function g if all calls to f are from within g. Block sinking does not approximate the returns of a function and does not change calls from tail to nontail or vice versa. Nesting f within g does not imply that f only expresses intraprocedural control flow within g and can share the same stack frame as g.

Contification is also superficially similar to Appel's loop headers [3]. However, once again, they are complementary. The introduction of a loop header is a transformation local to a particular function that allows self tail calls to be turned into jumps and loop invariant code to be moved into the header. It is a useful local optimization, which can take advantage of contification, but does not expose any new intraprocedural contro-flow information. Appel relies on inlining to do that. MLton introduces loop headers as one of the many FOL optimizations.

4.4 Well-definedness of the transformation

We state but do not prove in this paper that for safe analyses, the result of the transformation obeys the lexical scoping rules and type system of FOL. In this section, we focus on the more fundamental issue of showing that the transformation is well-defined, again, only for safe analyses. The rules in Figure 3 are not defined by induction on the structure of expressions, since the mutually recursive calls of $\mathcal{E}$ and

$\mathcal{F}$ use sets of functions determined by the analysis. Consequently, for some nonsensical analyses, the transformation is not well-defined.

As an example of a nonsensical analysis, suppose an analysis $\mathcal{A}$ specifies $\mathcal{A}(f) = g$ and $\mathcal{A}(g) = f$. This would force f to be contified in g and g to be contified in f, causing $\mathcal{F}$ and $\mathcal{E}$ to be undefined. The problem with such an analysis is that there is a "cycle" from f to g back to f through $\mathcal{A}$. In order to rule out such analyses, we define a directed graph $G = (\textsc{Node}, \textsc{Edge})$ based on analysis $\mathcal{A}$.

$$\begin{array}{lcl} \textsc{Node} & = & \mathit{Return} \\ \textsc{Edge} & = & \{(\mathcal{A}(f), f) \mid f \in \mathit{Func}\} \cup \\ & & \{(\mathcal{D}(k), k) \mid k \in \mathit{Cont}\} \end{array}$$

The mutually recursive calls to $\mathcal{F}$ and $\mathcal{E}$ in Figure 3 correspond to a traversal of G starting at $\mathsf{Unknown}$. Thus, if there are no cycles reachable from $\mathsf{Unknown}$ in G, the transformation is well-defined. This leads us to introduce the definition of an *acyclic* analysis, and to prove that a safe analysis is always acyclic, and hence always leads to a well-defined transformation.

Definition 3. An analysis $\mathcal{A}$ is *cyclic* if there exists a sequence $l_0, ..., l_n \in \mathit{Cont} \cup \mathit{Func}$ such that $l_0 = l_n$ and for all $0 \leq i < n$, either (1) $l_i \in \mathit{Func}$ and $\mathcal{A}(l_i) = l_{i+1}$ or (2) $l_i \in \mathit{Cont}$ and $\mathcal{D}(l_i) = l_{i+1}$. An analysis is *acyclic* if it is not cyclic.

The key to proving that safety implies acyclicity is the observation is that any purported cycle in a safe analysis must be composed of unreachable functions, which must be marked $\mathsf{Uncalled}$ by $*_1$, contradicting the definition of a cycle. We formalize this reasoning in the following theorem.

THEOREM 2. *If $\mathcal{A}$ is a safe analysis, then $\mathcal{A}$ is acyclic.*

PROOF. Suppose, by way of contradiction, that there exists $l_0, ..., l_n \in \mathit{Cont} \cup \mathit{Func}$ such that $l_0 = l_n$ and for each $0 \leq i < n$, either (1) $l_i \in \mathit{Func}$ and $\mathcal{A}(l_i) = l_{i+1}$ or (2) $l_i \in \mathit{Cont}$ and $\mathcal{D}(l_i) = l_{i+1}$. Condition (2) implies that there exists $l_i \in \mathit{Func}$. Condition (1) implies that for each $l_i \in \mathit{Func}$, $\mathcal{A}(l_i) \neq \mathsf{Uncalled}$. Hence $\mathcal{R}(l_i)$, by Lemma 1, and there exists a path of calls from f_{m} to l_i.

Consider a fixed path p from f_{m} to l_i, composed of nontail calls $(g_j, g_{j+1}, k_j) \in \mathrm{N}$ and tail calls $(g_j, g_{j+1}) \in \mathrm{T}$. There is a least j such that $g_j \in \{l_0, \ldots, l_n\}$, say $g_j = l_{i'}$.

We show that $\mathcal{A}(g_{j'}) \neq l_{i'+1}$ for all $j' \leq j$ by induction on j'. If $j' = 0$, then $g_{j'} = f_{\mathsf{m}}$ and $\mathcal{A}(g_{j'}) = \mathsf{Unknown}$ by $*_2$, because $\mathcal{A}$ is safe. If $j' > 0$, then either $(g_{j'-1}, g_{j'}, k_{j'-1}) \in \mathrm{N}$ or $(g_{j'-1}, g_{j'}) \in \mathrm{T}$ is on p. In the first case, $\mathcal{A}(g_{j'}) \in \{k_{j'-1}, \mathsf{Unknown}\}$ by $*_3$, because $\mathcal{A}$ is safe. By the minimality of j and condition (2), $k_{j'-1} \neq l_{i'+1}$. Hence, $\mathcal{A}(g_{j'}) \neq l_{i'+1}$. In the second case $\mathcal{A}(g_{j'}) \in \{g_{j'-1}, \mathcal{A}(g_{j'-1}), \mathsf{Unknown}\}$ by $*_4$, because $\mathcal{A}$ is safe. By the minimality of j and condition (1), $g_{j'-1} \neq l_{i'+1}$ and, by the induction hypothesis, $\mathcal{A}(g_{j'-1}) \neq l_{i'+1}$. Hence, $\mathcal{A}(g_{j'}) \neq l_{i'+1}$.

Thus $\mathcal{A}(l_{i'}) \neq l_{i'+1}$, which is a contradiction. □

Theorem 2 guarantees that G is an acyclic graph if $\mathcal{A}$ is safe. In fact, G is a forest of exactly two trees, rooted at $\mathsf{Uncalled}$ and $\mathsf{Unknown}$ respectively.

$$\mathcal{A}_{\mathrm{Call}}(f) = \begin{cases} \mathsf{Uncalled} & \text{if } \neg\mathcal{R}(f) \\ \mathsf{Unknown} & \text{if } f = f_{\mathsf{m}} \\ l & \text{if } \mathit{Outer}_{\mathrm{N}}(f) \cup \mathit{Outer}_{\mathrm{T}}(f) = \{l\} \\ & \quad \text{and } \mathit{Inner}_{\mathrm{N}}(f) = \emptyset \\ \mathsf{Unknown} & \text{otherwise} \end{cases}$$

$$\begin{array}{rcl} \mathit{Outer}_{\mathrm{N}}(f) & = & \{k \mid (g, f, k) \in \mathrm{N} \text{ and } \mathcal{R}(g) \text{ and } g \neq f\} \\ \mathit{Outer}_{\mathrm{T}}(f) & = & \{g \mid (g, f) \in \mathrm{T} \text{ and } \mathcal{R}(g) \text{ and } g \neq f\} \\ \mathit{Inner}_{\mathrm{N}}(f) & = & \{k \mid (f, f, k) \in \mathrm{N}\} \end{array}$$

Figure 4: Call analysis

5. ANALYSES

In this section, we present three safe analyses: the original contification analysis used in MLton, the analysis used in Moby, and our new dominator analysis. These analyses vary in their complexity and their utility in guiding transformations. Their range demonstrates the generality of our framework and the ease with which analyses can be defined on FOL. As in Section 4, we assume a fixed, but arbitrary, program $P = (f_{\mathsf{m}}, \mathrm{N}, \mathrm{T})$ when defining the analyses.

5.1 The $\mathcal{A}_{\mathrm{Call}}$ Analysis

Our first analysis, the *call analysis*, is a syntax driven analysis that has been used in MLton since September 1998. The analysis is based on the observation that a function has one return location if there is exactly one reachable call to the function from outside its body and if there are only tail calls to the function within its body. For example, this was the case with the `loop` function used to sum the elements in a vector in Section 3.

We formally define the call analysis in Figure 4. We define the multisets $\mathit{Outer}_{\mathrm{N}}(f)$, $\mathit{Outer}_{\mathrm{T}}(f)$, and $\mathit{Inner}_{\mathrm{N}}(f)$, corresponding to the continuations of reachable nontail calls to f from outside its body, the reachable tail callers of f from outside its body, and the continuations of nontail calls to f from inside its body, respectively. For a function f, there is one reachable call from outside its body if and only if $\mathit{Outer}_{\mathrm{N}}(f) \cup \mathit{Outer}_{\mathrm{T}}(f) = \{l\}$. Further, all calls from f to itself are in tail position if and only if $\mathit{Inner}_{\mathrm{N}}(f) = \emptyset$.

The proof of safety of the call analysis is straightforward.

THEOREM 3. *$\mathcal{A}_{\mathrm{Call}}$ is safe.*

PROOF. We show that $\mathcal{A}_{\mathrm{Call}}$ satisfies each of the safety conditions.

$*_1$ If $\neg\mathcal{R}(f)$, then $\mathcal{A}_{\mathrm{Call}}(f) = \mathsf{Uncalled}$ by the first clause.

$*_2$ $\mathcal{A}_{\mathrm{Call}}(f_{\mathsf{m}}) = \mathsf{Unknown}$ by the second clause.

$*_3$ Suppose $(f, g, k) \in \mathrm{N}$ such that $\mathcal{R}(f)$. Therefore, $\mathcal{R}(g)$ and $\mathcal{A}_{\mathrm{Call}}(g) \neq \mathsf{Uncalled}$, because the first clause does not apply. If $f = g$, then $\mathit{Inner}_{\mathrm{N}}(g) \neq \emptyset$ and $\mathcal{A}_{\mathrm{Call}}(f) = \mathsf{Unknown} \in \{k, \mathsf{Unknown}\}$. If $f \neq g$, then $\mathit{Outer}_{\mathrm{N}}(g) \supseteq \{k\}$ and $\mathcal{A}_{\mathrm{Call}}(g) \in \{k, \mathsf{Unknown}\}$.

$*_4$ Suppose $(f, g) \in \mathrm{T}$ such that $f \neq g$ and $\mathcal{R}(f)$. Therefore, $\mathcal{R}(g)$ and $\mathcal{A}_{\mathrm{Call}}(g) \neq \mathsf{Uncalled}$, because the first clause does not apply. Also, $\mathit{Outer}_{\mathrm{T}}(g) \supseteq \{f\}$ and $\mathcal{A}_{\mathrm{Call}}(g) \in \{f, \mathsf{Unknown}\} \subseteq \{f, \mathcal{A}_{\mathrm{Call}}(f), \mathsf{Unknown}\}$. □

Although the call analysis is useful in practice, it fails to contify functions in many simple cases. For example, consider the following program fragment.

$$\mathcal{A}_{\text{Cont}} = \text{lfp}(C)$$

$$
\begin{aligned}
& C : \textit{Analysis} \rightarrow \textit{Analysis} \\
& C(A)(f) = \text{if } f = f_{\mathsf{m}} \\
& \qquad \text{then } \mathsf{Unknown} \\
& \qquad \text{else } \bigsqcup \begin{pmatrix} \{k \mid (g,f,k) \in \text{N and } \mathcal{R}(g)\} \\ \cup \{\mathcal{A}(g) \mid (g,f) \in \text{T and } \mathcal{R}(g)\} \end{pmatrix}
\end{aligned}
$$

Figure 5: Continuation analysis

```
fun fm () = ... k (f ()) ... k (g ()) ...
fun f () = ... g () ... h () ...
fun g () = ... f () ... h () ...
fun h () = ...
```

In this case, `f`, `g`, and `h` always return to the continuation `k`, either directly or by returning to a function which always returns to the continuation `k`. Thus we could contify `f`, `g`, and `h` within `fm`. However, because each function is called at multiple places outside its body, the call analysis is useless: $\mathcal{A}_{\text{Call}}(\texttt{f}) = \mathcal{A}_{\text{Call}}(\texttt{g}) = \mathcal{A}_{\text{Call}}(\texttt{h}) = \mathsf{Unknown}$. We would like an algorithm to compute the safe analysis $\mathcal{A}(\texttt{f}) = \mathcal{A}(\texttt{g}) = \mathcal{A}(\texttt{h}) = \texttt{k}$. The analysis in the next section will do just that.

5.2 The $\mathcal{A}_{\text{Cont}}$ Analysis

Our second analysis, the *continuation analysis*, is based on the analysis in Moby, which "computes an approximation of return continuations of each known function" [21]. Unlike the call analysis, which gives up if a function is called from many places, the continuation analysis contifies a function if the function returns to a single continuation through one or more (possibly disjoint) sequences of tail calls. Viewed from the point of view of CPS, $\mathcal{A}_{\text{Cont}}$ determines when a continuation variable takes on a constant value.

To define the continuation analysis, we arrange the elements of *Return* in a lattice. We define $\mathsf{Uncalled} \sqsubseteq l \sqsubseteq \mathsf{Unknown}$ for any $l \in \textit{Cont} \cup \textit{Func}$ and define $\rho_1 \sqcup \rho_2$ to be the least upper bound of ρ_1 and ρ_2. We extend $\sqsubseteq$ and $\sqcup$ pointwise to form a lattice on *Analysis*.

The continuation analysis is defined via the least fixpoint in Figure 5. The idea behind the analysis is that a function f returns to continuation k if all reachable nontail calls to f use k and if all tail callers of f also return to continuation k. The least fixpoint ties the recursion in the previous sentence. It ensures that $\mathcal{A}_{\text{Cont}}(f) = k$ if and only if all paths of reachable tail calls to f start with a function that returns to continuation k.

THEOREM 4. *$\mathcal{A}_{\text{Cont}}$ is safe.*

PROOF. We show that $\mathcal{A}_{\text{Cont}}$ satisfies each of the safety conditions.

$*_1$ Recall $\mathcal{A}_{\text{Triv}}$ defined in Section 4.1. If $\mathcal{R}(f)$, then $C(\mathcal{A}_{\text{Triv}})(f) \sqsubseteq \mathsf{Unknown} = \mathcal{A}_{\text{Triv}}(f)$. If $\neg\mathcal{R}(f)$, then $C(\mathcal{A}_{\text{Triv}})(f) = \mathsf{Uncalled} = \mathcal{A}_{\text{Triv}}(f)$. Hence, we have $C(\mathcal{A}_{\text{Triv}}) \sqsubseteq \mathcal{A}_{\text{Triv}}$ and $\mathcal{A}_{\text{Cont}} = \text{lfp}(C) \sqsubseteq \mathcal{A}_{\text{Triv}}$. Therefore, if $\neg\mathcal{R}(f)$, then $\mathcal{A}_{\text{Cont}}(f) \sqsubseteq \mathcal{A}_{\text{Triv}}(f) = \mathsf{Uncalled}$. Therefore, $\mathcal{A}_{\text{Cont}}(f) = \mathsf{Uncalled}$.

$*_2$
$$
\begin{aligned}
\mathcal{A}_{\text{Cont}}(f_{\mathsf{m}}) &= C(\mathcal{A}_{\text{Cont}})(f_{\mathsf{m}}) \\
&= \text{if } f = f_{\mathsf{m}} \text{ then } \mathsf{Unknown} \text{ else } \ldots \\
&= \mathsf{Unknown}.
\end{aligned}
$$

$*_3$ Suppose $(f,g,k) \in \text{N}$ such that $\mathcal{R}(f)$.

$$
\begin{aligned}
& \mathcal{A}_{\text{Cont}}(g) \\
& = C(\mathcal{A}_{\text{Cont}})(g) \\
& = \text{if } g = f_{\mathsf{m}} \\
& \quad \text{then } \mathsf{Unknown} \\
& \quad \text{else } \bigsqcup \begin{pmatrix} \{k \mid (f,g,k) \in \text{N and } \mathcal{R}(f)\} \\ \cup \{\mathcal{A}_{\text{Cont}}(f) \mid (f,g) \in \text{T and } \mathcal{R}(f)\} \end{pmatrix} \\
& = \text{if } g = f_{\mathsf{m}} \text{ then } \mathsf{Unknown} \text{ else } \sqcup(\{k\} \cup \ldots) \\
& \in \{k, \mathsf{Unknown}\}.
\end{aligned}
$$

$*_4$ Suppose $(f,g) \in \text{T}$ such that $\mathcal{R}(f)$. Then

$$
\begin{aligned}
& \mathcal{A}_{\text{Cont}}(g) \\
& = C(\mathcal{A}_{\text{Cont}})(g) \\
& = \text{if } g = f_{\mathsf{m}} \\
& \quad \text{then } \mathsf{Unknown} \\
& \quad \text{else } \bigsqcup \begin{pmatrix} \{k \mid (f,g,k) \in \text{N and } \mathcal{R}(f)\} \\ \cup \{\mathcal{A}_{\text{Cont}}(f) \mid (f,g) \in \text{T and } \mathcal{R}(f)\} \end{pmatrix} \\
& = \text{if } g = f_{\mathsf{m}} \text{ then } \mathsf{Unknown} \text{ else } \sqcup(\mathcal{A}_{\text{Cont}}(f) \cup \ldots) \\
& \in \{\mathcal{A}_{\text{Cont}}(f), \mathsf{Unknown}\} \\
& \subseteq \{f, \mathcal{A}_{\text{Cont}}(f), \mathsf{Unknown}\}. \quad \square
\end{aligned}
$$

The continuation analysis differs from the analysis in [21] in several ways. First, our analysis operates over the entire program, while Reppy's analyzes a module in isolation. Second, ours operates over a first-order IL, while Reppy's operates over a higher-order language. Third, ours runs after MLton's control-flow analysis, which exposes control-flow information in the first order FOL program. Reppy's analysis introduces imprecision when escaping function cause unknown control-flow. Finally, since our language is already in continuation-passing style, there is no need for "local CPS conversion" in order to apply the contification transformation.

The example at the end of Section 5.1 demonstrates that on some programs $\mathcal{A}_{\text{Cont}}$ contifies more functions than $\mathcal{A}_{\text{Call}}$. One can also construct programs on which $\mathcal{A}_{\text{Call}}$ will contify more functions than $\mathcal{A}_{\text{Cont}}$. Furthermore, there are programs with contifiable functions that are found by neither analysis. For example, consider the following program fragment.

```
fun fm () = ... k1 (f ()) ... k2 (f ()) ...
fun f () = ... g1 () ... g2 () ...
fun g1 () = ... h () ...
fun g2 () = ... h () ...
fun h () = ...
```

The following three analyses, $\mathcal{A}_{\text{Call}}$, $\mathcal{A}_{\text{Cont}}$, and $\mathcal{A}$, are safe.

	$\mathcal{A}_{\text{Call}}$	$\mathcal{A}_{\text{Cont}}$	$\mathcal{A}$
`fm`	Unknown	Unknown	Unknown
`f`	Unknown	Unknown	Unknown
`g1`	`f`	Unknown	`f`
`g2`	`f`	Unknown	`f`
`h`	Unknown	Unknown	`f`

The analysis $\mathcal{A}$ captures our intuition that `h` can be contified along with `g1` and `g2` within `f`. The call analysis fails to contify `h` because it is called from more than one place. Further, although `h` always returns to `f`, the continuation analysis fails to contify `h` because it determines that `f`, `g1`, `g2`, and `h` all have the same set of multiple return locations, $\{\texttt{k1}, \texttt{k2}\}$. The analysis in the next section will compute $\mathcal{A}$.

$$
\begin{aligned}
\mathcal{A}_{\text{Dom}}(f) = {} & \text{if } \mathit{parent}_D(f) = \mathsf{Root} \\
& \quad \text{then if } \mathcal{R}(f) \\
& \qquad \text{then } \mathsf{Unknown} \\
& \qquad \text{else } \mathsf{Uncalled} \\
& \quad \text{else } l, \text{ where } l \in \mathit{ancestors}_D(f) \text{ and} \\
& \qquad \mathit{parent}_D(l) = \mathsf{Root}
\end{aligned}
$$

D is the dominator tree of $G = (\textsc{Node}, \textsc{Edge})$

$$
\begin{array}{lcll}
\textsc{Node} & = & \{\mathsf{Root}\} \cup \mathit{Cont} \cup \mathit{Func} & \\
\textsc{Edge} & = & \{(\mathsf{Root}, f_{\mathsf{m}})\} \cup & \dagger_1 \\
& & \{(\mathsf{Root}, k) \mid k \in \mathit{Cont}\} \cup & \dagger_2 \\
& & \{(\mathsf{Root}, f) \mid \neg\mathcal{R}(f)\} \cup & \dagger_3 \\
& & \{(f, g) \mid (f, g) \in \mathrm{T} \text{ and } \mathcal{R}(f)\} \cup & \dagger_4 \\
& & \{(k, g) \mid (f, g, k) \in \mathrm{N} \text{ and } \mathcal{R}(f)\} & \dagger_5
\end{array}
$$

Figure 6: Dominator analysis

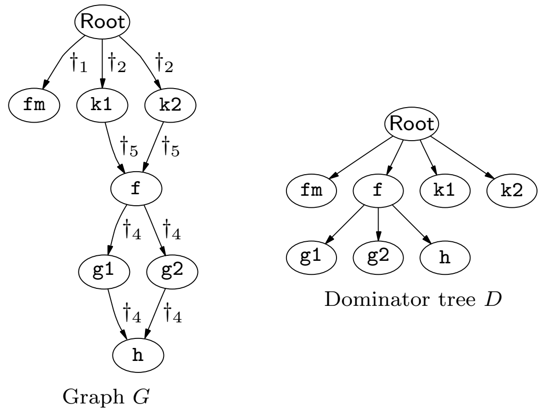

Figure 7: Dominator example

5.3 The $\mathcal{A}_{\text{Dom}}$ Analysis

Our final analysis, the *dominator analysis*, fully utilizes the control flow information available in FOL to determine exactly how far a function should be allowed to return through tail calls. The analysis can contify a function f in some cases even if f's continuation is not constant. After defining the dominator analysis, we will prove that it is both safe and maximal. That is, it contifies at least as many functions as $\mathcal{A}_{\text{Call}}$, $\mathcal{A}_{\text{Cont}}$, or any other safe analysis.

The dominator analysis, $\mathcal{A}_{\text{Dom}}$, is defined in Figure 6. It uses a directed graph G similar to the call graph of the program, but which contains the return information needed for contification. For $f \in \mathit{Func}$, each edge $(l, f) \in \textsc{Edge}$ indicates that f returns to the location l. If $l = \mathsf{Root}$, then the edge indicates that f has no return location; either f is uncalled or f is the main function. The dominator analysis is defined using the dominator tree D of G. As a reminder, a node n *dominates* a node n' in a graph if and only if every path from the root to n' goes through n [17]. Further, node n *immediately dominates* node n' if n dominates n' and every dominator of n' (other than n') dominates n. The nodes of the graph can be arranged into the *dominator tree*, D, where n is the parent of n' if n is the immediate dominator of n'. Figure 7 shows the graph G and the dominator tree D corresponding to the program fragment at the end of Section 5.2.

To ensure that the dominator tree of G exists, we need the following lemma, which shows that G is a connected graph rooted at Root.

LEMMA 5. *For all $l \in \mathit{Cont} \cup \mathit{Func}$, there is a path from Root to l in G.*

PROOF. If $l \in \mathit{Cont}$, then $(\mathsf{Root}, l) \in \textsc{Edge}$ by $\dagger_2$. If $l \in \mathit{Func}$ and $\neg\mathcal{R}(l)$, then $(\mathsf{Root}, l) \in \textsc{Edge}$ by $\dagger_3$. Finally, if $l \in \mathit{Func}$ and $\mathcal{R}(l)$, then there is a path of calls from f_{m} to l. We proceed by induction on the length of the path. If $n = 0$ then $l = f_{\mathsf{m}}$ and $(\mathsf{Root}, l) \in \textsc{Edge}$ by $\dagger_1$. If $n > 0$, then there exists $f \in \mathit{Func}$ such that there exists a path of $n - 1$ calls from f_{m} to f and $(f, l, k) \in \mathrm{N}$ or $(f, l) \in \mathrm{T}$. If $(f, l, k) \in \mathrm{N}$, then $(k, l) \in \textsc{Edge}$ by $\dagger_5$ and $(\mathsf{Root}, k) \in \textsc{Edge}$ by $\dagger_2$. If $(f, l) \in \mathrm{T}$, then $(f, l) \in \textsc{Edge}$ by $\dagger_4$ and there is a path from Root to f in G by the induction hypothesis. $\square$

The set of dominators of a node $f \in \mathit{Func}$ is the set of locations to which f always returns in any execution of the program. However, we cannot define $\mathcal{A}_{\text{Dom}}(f)$ to be an arbitrary dominator of f; this could easily lead to an unsafe analysis. Instead, for $l \in \mathit{Cont} \cup \mathit{Func}$, let $\mathit{parent}_D(l)$ be the parent of l in D and let $\mathit{ancestors}_D(l)$ be the set of ancestors of l in D. The dominator analysis defines $\mathcal{A}_{\text{Dom}}(f)$ as the dominator of f closest to Root. Thus, we see in Figure 7 that $\mathcal{A}_{\text{Dom}}(\mathtt{g1}) = \mathcal{A}_{\text{Dom}}(\mathtt{g2}) = \mathcal{A}_{\text{Dom}}(\mathtt{h}) = \mathtt{f}$.

THEOREM 6. *$\mathcal{A}_{\text{Dom}}$ is safe.*

PROOF. We show that $\mathcal{A}_{\text{Dom}}$ satisfies each of the safety conditions.

- $*_1$ If $\neg\mathcal{R}(f)$, then $(\mathsf{Root}, f) \in \textsc{Edge}$ by $\dagger_3$. Hence, $\mathit{parent}_D(f) = \mathsf{Root}$ and $\mathcal{A}_{\text{Dom}}(f) = \mathsf{Uncalled}$.
- $*_2$ Note $(\mathsf{Root}, f_{\mathsf{m}}) \in \textsc{Edge}$ by $\dagger_1$ and $\mathcal{R}(f_{\mathsf{m}})$. Hence, $\mathit{parent}_D(f_{\mathsf{m}}) = \mathsf{Root}$ and $\mathcal{A}_{\text{Dom}}(f_{\mathsf{m}}) = \mathsf{Unknown}$.
- $*_3$ Suppose $(f, g, k) \in \mathrm{N}$ and $\mathcal{R}(f)$. Hence, $\mathcal{R}(g)$, $(\mathsf{Root}, k) \in \textsc{Edge}$ by $\dagger_2$, and $(k, g) \in \textsc{Edge}$ by $\dagger_5$. Therefore, $\mathit{parent}_D(g) \in \{\mathsf{Root}, k\}$ and $\mathit{parent}_D(k) = \mathsf{Root}$. Hence, $\mathcal{A}_{\text{Dom}}(g) \in \{k, \mathsf{Unknown}\}$.
- $*_4$ Suppose $(f, g) \in \mathrm{T}$ and $\mathcal{R}(f)$. Hence, $\mathcal{R}(g)$ and $(f, g) \in \mathit{Edge}$ by $\dagger_4$. If $\mathit{parent}_D(g) = \mathsf{Root}$, then $\mathcal{A}_{\text{Dom}}(g) = \mathsf{Unknown} \in \{f, \mathcal{A}_{\text{Dom}}(f), \mathsf{Unknown}\}$. If $\mathit{parent}_D(g) \neq \mathsf{Root}$, then let $l \in \mathit{ancestors}_D(g)$ such that $\mathit{parent}_D(l) = \mathsf{Root}$. Thus, $\mathcal{A}_{\text{Dom}}(g) = l$. If $l = f$, then $\mathcal{A}_{\text{Dom}}(g) = f \in \{f, \mathcal{A}_{\text{Dom}}(f), \mathsf{Unknown}\}$. If $l \neq f$, then $l \in \mathit{ancestors}_D(f)$ and $\mathit{parent}_D(f) \neq \mathsf{Root}$, because l dominates f in G. Therefore, $\mathcal{A}_{\text{Dom}}(f) = l$ and $\mathcal{A}_{\text{Dom}}(g) = \mathcal{A}_{\text{Dom}}(f) \in \{f, \mathcal{A}_{\text{Dom}}(f), \mathsf{Unknown}\}$. $\square$

To show that $\mathcal{A}_{\text{Dom}}$ is maximal, we need the following lemma that relates a safe analysis to paths in the graph G.

LEMMA 7. *Let $\mathcal{A}$ be safe and let $\mathsf{Root}, l, f_0, \ldots, f_n$ be a path in G. Then $\mathcal{A}(f_n) \in \{f_{n-1}, \ldots, f_0, l, \mathsf{Unknown}\}$.*

PROOF. By induction on the length of the path.

$n = 0$: Suppose $l \in \mathit{Func}$. First, note that $(\mathsf{Root}, l) \in \textsc{Edge}$ either by $\dagger_1$ (with $l = f_{\mathsf{m}}$) or by $\dagger_3$ (with $\neg\mathcal{R}(l)$). Second, note that $(l, f_0) \in \textsc{Edge}$ by $\dagger_4$ (with $(l, f_0) \in \mathrm{T}$ and $\mathcal{R}(l)$). Thus, $l = f_{\mathsf{m}}$. Therefore, $\mathcal{A}(l) = \mathsf{Unknown}$ by $*_2$ and $\mathcal{A}(f_0) \in \{l, \mathcal{A}(l), \mathsf{Unknown}\} = \{l, \mathsf{Unknown}\}$, by $*_4$.

Program	Lines	Contify time (seconds)	Compile time (seconds)	Executable sizes (bytes, normalized) none	$\mathcal{A}_{\text{Call}}$	$\mathcal{A}_{\text{Cont}}$	$\mathcal{A}_{\text{Dom}}$	Run times (normalized) $\mathcal{A}_{\text{Call}}$	$\mathcal{A}_{\text{Cont}}$	$\mathcal{A}_{\text{Dom}}$
`barnes-hut`	1,262	0.06	4.68	67,641	0.93	0.93	0.94	0.68	0.97	0.81
`count-graphs`	538	0.05	2.82	66,012	0.85	0.83	0.83	0.73	0.80	0.71
`hamlet`	22,895	2.54	103.59	1,107,661	0.95	0.92	0.90	0.79	0.86	0.94
`kit`	73,490	13.24	682.18	5,729,035	1.02	0.99	0.97	0.72	0.93	0.72
`lexgen`	1,327	0.28	10.08	170,578	0.89	0.90	0.85	0.73	0.76	0.68
`mlton`	92,134	13.26	572.55	5,242,115	0.94	0.91	0.90	0.72	0.88	0.74
`mlyacc`	7,295	0.67	37.91	472,514	0.96	0.94	0.93	0.59	0.91	0.59
`raytrace`	2,378	0.36	19.89	235,058	0.89	0.93	0.84	1.05	1.05	1.05
`tensor`	2,947	0.17	5.42	75,418	0.88	0.87	0.90	0.14	0.14	0.15
`vliw`	3,694	0.65	24.20	278,006	1.03	0.97	1.00	0.59	0.81	0.59
`zern`	595	0.02	1.49	45,375	0.89	0.89	0.91	0.29	0.29	0.31

Table 1: Compilation and run time statistics

Suppose $l \in Cont$. Note that $(\mathsf{Root}, k) \in \text{EDGE}$ by $\dagger_2$ and $(l, f_0) \in \text{EDGE}$ by $\dagger_4$ (with $(g, f_0, l) \in \text{N}$ and $\mathcal{R}(g)$). Therefore, $\mathcal{A}(f_0) \in \{l, \mathsf{Unknown}\}$, by $*_3$.

$n > 0$: Note that $(f_{n-1}, f_n) \in \text{EDGE}$ by $\dagger_4$ with $(f_{n-1}, f_n) \in \text{T}$ and $\mathcal{R}(f_{n-1})$. By $*_4$, $\mathcal{A}(f_n) \in \{f_{n-1}, \mathcal{A}(f_{n-1}), \mathsf{Unknown}\}$. From the induction hypothesis, we know that $\mathcal{A}(f_{n-1}) \in \{f_{n-2}, \ldots, f_0, l, \mathsf{Unknown}\}$. Thus we know that $\mathcal{A}(f_n) \in \{f_{n-1}, f_{n-2}, \ldots, f_0, l, \mathsf{Unknown}\}$. $\square$

A simple corollary of Lemma 7 is the key to proving that $\mathcal{A}_{\text{Dom}}$ is maximal.

COROLLARY 8. *If $\mathcal{A}$ is safe and $\mathcal{A}(f) \in Cont \cup Func$ then $\mathcal{A}(f)$ dominates f in G.*

THEOREM 9. *$\mathcal{A}_{\text{Dom}}$ is maximal.*

PROOF. Let $\mathcal{B}$ be an arbitrary safe analysis and $f \in Func$ be arbitrary. We consider the possible values of $\mathcal{B}(f)$. If $\mathcal{B}(f) = \mathsf{Unknown}$, we are done. If $\mathcal{B}(f) = \mathsf{Uncalled}$, then $\neg\mathcal{R}(f)$ and $\mathcal{A}_{\text{Dom}}(f) = \mathsf{Uncalled}$ by Lemma 1, because $\mathcal{B}$ and $\mathcal{A}_{\text{Dom}}$ are safe analyses. Finally, suppose $\mathcal{B}(f) = l \in Cont \cup Func$. Hence, $f \neq f_{\mathsf{m}}$ and $\mathcal{R}(f)$ by $*_2$ and Lemma 1, because $\mathcal{B}$ is safe. By examining the cases in the definition of EDGE, we see that every path from Root to f in G has length greater than 1. By Corollary 8 applied to $\mathcal{B}$, $l = B(f)$ dominates f in G. Therefore, $parent_D(f) \neq \mathsf{Root}$ and $\mathcal{A}_{\text{Dom}}(f) \in Cont \cup Func$. $\square$

The argument above proves that $\mathcal{A}_{\text{Dom}}$ is a maximal analysis, but we can be more precise about the relationship between $\mathcal{A}_{\text{Dom}}$ and other safe analyses. One can extend the argument above to prove the following theorem.

THEOREM 10. *Let $\mathcal{A}$ be safe. If $\mathcal{A}(f) = k \in Cont$, then $\mathcal{A}_{\text{Dom}}(f) = k$. If $\mathcal{A}(f) \in Func$, then $\mathcal{A}_{\text{Dom}}(f) \in Cont \cup Func$.*

This theorem shows that the dominator analysis favors contification at continuations over contification in functions. It also implies that the contification and dominator analyses should agree on many functions. In fact, k dominates f in the graph defined in Figure 6 if and only if all paths of reachable tail calls start with a function that returns to k; so, $\mathcal{A}_{\text{Cont}}$ is equivalent to an analysis that assigns $\mathcal{A}(f) = k$ if k dominates f.

6. EXPERIMENTS

As described in Section 1, many compiler optimizations are enabled by exposing the intraprocedural control-flow of a program through the contification transformation. It is therefore not surprising that a contification pass (similar to the call analysis) was the first FOL optimization added to MLton in September 1998. The contification pass now runs at three places in the FOL optimizer, with intervening optimization passes including constant propagation, dead-code elimination, inlining, raise-to-jump transformation, loop optimizations, and shrink reductions [5].

To demonstrate the practicality and benefits of the transformation and analyses described in this paper, we implemented a new contification pass in the MLton compiler. Paralleling the presentation in Section 4, in which we separate contification into analysis and transformation, our new pass consists of an analysis phase that produces an annotation and a transformation phase that contifies based on the annotation. We have implemented each of the analyses described in Section 5. The implementations of $\mathcal{A}_{\text{Call}}$ and $\mathcal{A}_{\text{Cont}}$ are straightforward. The implementation of $\mathcal{A}_{\text{Dom}}$ uses the Lengauer-Tarjan dominator algorithm [17] as presented in [19].

We measured the impact of contification on compile times and running times for a representative sample of benchmarks with sizes up to 92K lines. Among the benchmarks, `lexgen`, `mlyacc`, and `vliw` are standard [2]; `barnes-hut`, `tensor`, and `zern` are floating-point intensive and `count-graphs` is mostly symbolic.[1] The `raytrace` benchmark was the winning entry in the Third Annual ICFP Programming Contest.[2] The `mlton` benchmark is the compiler itself; `kit` is the ML-Kit (Version 3) [23]; `hamlet` is the HaMLet SML interpreter.[3] The benchmarks were compiled with the native x86 backend and executed on an 800 MHz Intel Pentium III

[1] Juan Jose Garcia Ripoll (`worm@arrakis.es`) wrote `tensor`, David McClain (`dmcclain@azstarnet.com`) wrote `zern`, and Henry Cejtin (`henry@sourcelight.com`) wrote `count-graphs`.

[2] Team PLClub (`http://www.cis.upenn.edu/~sumii/icfp`) wrote the original version of `raytrace` in O'Caml. Stephen Weeks translated it to SML, and John Reppy (`jhr@research.bell-labs.com`) made further modifications.

[3] Andreas Rossberg (`rossberg@ps.uni-sb.de`) wrote `hamlet`.

Analysis	barnes-hut			count-graphs			hamlet			kit		
none	0/118	0/96	0/43	0/102	0/82	0/48	0/1689	0/1625	0/943	0/7381	0/7192	0/3083
$\mathcal{A}_{\text{Call}}$	67/118	6/49	0/7	61/102	7/41	0/11	384/1689	14/1273	2/648	2776/7381	69/4527	2/1691
$\mathcal{A}_{\text{Cont}}$	64/118	6/52	0/10	62/102	8/40	0/10	370/1689	13/1287	0/702	2570/7381	54/4736	3/1900
$\mathcal{A}_{\text{Dom}}$	72/118	5/44	0/9	69/102	6/33	0/9	653/1689	12/1004	0/578	3402/7381	50/3911	3/1585

Analysis	lexgen			mlton			mlyacc			raytrace		
none	0/245	0/229	0/125	0/9921	0/8435	0/2389	0/818	0/779	0/453	0/301	0/277	0/121
$\mathcal{A}_{\text{Call}}$	121/245	3/124	0/37	4207/9921	439/4695	2/1153	419/818	9/396	1/111	124/301	10/177	0/39
$\mathcal{A}_{\text{Cont}}$	109/245	2/136	0/51	3656/9921	544/5247	0/1456	360/818	7/455	1/172	117/301	10/184	0/46
$\mathcal{A}_{\text{Dom}}$	139/245	2/106	0/34	4749/9921	438/4156	0/1049	473/818	7/342	0/103	172/301	9/129	0/32

Analysis	tensor			vliw			zern		
none	0/156	0/129	0/59	0/501	0/447	0/247	0/49	0/41	0/25
$\mathcal{A}_{\text{Call}}$	98/156	10/57	0/7	229/501	20/268	0/92	34/49	3/15	0/2
$\mathcal{A}_{\text{Cont}}$	94/156	9/61	0/11	209/501	15/288	0/115	36/49	1/13	0/2
$\mathcal{A}_{\text{Dom}}$	102/156	10/53	0/6	254/501	16/245	0/83	37/49	1/12	0/1

Table 2: Number of functions contified in each contification pass

with 256 MB of memory, with the exceptions of `mlton` and `kit`, which were compiled on a 733 MHz Intel Pentium III with 512 MB of memory. All benchmarks are available at `http://www.sourcelight.com/MLton`.

In the first part of Table 1, we report the number of lines of SML for each benchmark, the total amount of time spent in contification, and the total compile time in seconds. The number of lines does not include approximately 8000 lines of basis library code that MLton prefixes to each program. The contify time is the sum over all three contification passes, and includes the time to compute all three analyses and to perform the transformation based on $\mathcal{A}_{\text{Dom}}$. The total contification time was typically about 2% of the total compile time, and was never more than 4%.

In the second part of Table 1, we report the absolute size in bytes for each benchmark compiled with contification disabled. We also report the sizes when compiled using each analysis, normalized to the absolute size. The results show that contification almost always has a beneficial effect on executable size. In the third part of Table 1, we report the running time of each benchmark, compiled using each contification analysis, normalized to the running time with contification disabled. Unsurprisingly, these results show that contification typically yields a significant speedup.

In Table 2, we report the number of functions contified by each analysis. For each benchmark, there are four rows, one for each analysis (none means contification was disabled). Each row contains the counts for each of the three contification passes. Each cell in the table reports the number functions contified, followed by a "/", followed by the total number of functions in the program input to the contification pass.

The counts show that for most of the benchmarks, by the last round of contification, there are no contifiable functions detectable by the analysis. They also show that contification, by any analysis, has a significant impact on the total number of functions in the program. Although other optimizations reduce the number of functions (e.g., inlining), in these examples, a contified program generally has at most half the number of functions of an uncontified program.

The counts also show that in almost all cases, because $\mathcal{A}_{\text{Dom}}$ is maximal, it produces a result with fewer functions than $\mathcal{A}_{\text{Call}}$ or $\mathcal{A}_{\text{Cont}}$. The only exception is `barnes-hut`, where $\mathcal{A}_{\text{Dom}}$ yields a program with 9 functions, but $\mathcal{A}_{\text{Call}}$ yields a program with 7 functions. In this case, $\mathcal{A}_{\text{Dom}}$ contified a function that $\mathcal{A}_{\text{Call}}$ did not, increasing its size beyond the limit used by the inliner that runs between the second and third rounds of contification. This interaction with the inliner also explains why contification, which should enable more optimizations, does not always lead to smaller code size. It also partially explains why contification with $\mathcal{A}_{\text{Dom}}$ does not always lead to better running times than contification with $\mathcal{A}_{\text{Call}}$ and $\mathcal{A}_{\text{Cont}}$. We also suspect that this discrepancy in running times is an artifact of the evolution of the FOL optimizer, which developed around an original contification pass similar to $\mathcal{A}_{\text{Call}}$.

6.1 Mutual recursion in FOL

This section contains a minor note about a difference between the FOL language described in this paper and as implemented in MLton.

The contification transformation is complicated by the fact that MLton's current implementation of FOL does not allow continuations to be simultaneously declared as mutually recursive. Recall that the contification transformation described in Section 4.2 requires the set of functions $\{f \in \mathit{Func} \mid \mathcal{A}(f) = l\}$ to be declared simultaneously. The limitations of the implementation of FOL are problematic when this set contains multiple functions. However, because continuation declarations can be nested, it is possible to define mutually recursive continuations by nesting one within another (this approach is used in [9]). Unfortunately, this nesting is not sufficient to contify all sets of mutually recursive definitions. For example, we cannot contify `g1` and `g2` in the following code fragment.

```
fun f (b) = let cont k (x) = ...
                cont l1 () = k (g1 ())
                cont l2 () = k (g2 ())
            in if b then l1 () else l2 ()
            end
fun g1 () = ... g2 () ...
fun g2 () = ... g1 () ...
```

Fortunately, the limitations of FOL did not significantly affect the results in this section. When our full benchmark suite (including 18 additional benchmarks not reported here) is compiled with the dominator analysis, there are over 11,000 functions marked contifiable, only 78 of which must be nested and only 25 of which cannot be contified due to the absence of mutual recursion.

The absence of mutually recursive continuations in FOL is a historical accident, and is not fundamental. We are considering improving MLton to handle them.

7. CONCLUSION

Contification is not a new concept in functional-language compiler optimizations, but all previous work in this area has focused on presenting a single contification analysis and transformation. We have presented a simple, yet general, framework for expressing contification analyses. This generality has allowed us to define a maximality criterion for analyses and to introduce a single transformation that can be applied to any analysis satisfying a safety condition. We have shown how to express a number of existing analyses in our framework and presented a new maximal analysis based on the dominator tree of a program's call graph.

Finally, our implementation in MLton has shown that contification is efficient, taking a small percentage of compile time, and leads to improved run times. Although we have verified that the dominator analysis contifies more functions than existing analyses, we have not been able to show that this leads to consistently better run times. Nevertheless, the increased contification has convinced us to switch to the dominator analysis MLton. We believe that the undesirable interaction with inlining can be fixed and that the improved intraprocedural control-flow information provided by the dominator analysis will provide more benefits than other analyses to existing and planned optimizations.

ACKNOWLEDGEMENTS

We would like to thank John Reppy for providing helpful information about contification in Moby. We would like to thank Henry Cejtin, Neal Glew, Suresh Jagannathan, and the anonymous referees for helpful comments on the paper.

8. REFERENCES

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[2] A. W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[3] A. W. Appel. Loop headers in λ-calculus or CPS. *Lisp and Symbolic Computation*, 7:337–343, 1994.

[4] A. W. Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[5] A. W. Appel and T. Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, 1997.

[6] H. Cejtin, S. Jagannathan, and S. T. Weeks. Flow-directed closure conversion for typed languages. In *European Symposium on Programming*, pages 56–71, Mar. 2000.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[8] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[9] O. Danvy and U. P. Schultz. Lambda-dropping: Transforming recursive equations into programs with block structure. *Theoretical Computer Science*, 248(1–2):243–287, 2000.

[10] M. Elsman. Static interpretation of modules. In *International Conference on Functional Programming*, pages 208–219, Sept. 1999.

[11] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Conference on Programming Language Design and Implementation*, pages 237–247, June 1993.

[12] R. Harper and J. C. Mitchell. On the type structure of Standard ML. *Transactions on Programming Languages and Systems*, 15(2):211–252, April 1993.

[13] N. D. Jones. Flow analysis of lambda expressions. In *International Colloquium on Automata, Languages, and Programming*, volume 115, pages 114–128. Springer-Verlag, 1981.

[14] R. A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Workshop on Intermediate Representations*, pages 13–22, Jan. 1995.

[15] R. A. Kelsey and P. Hudak. Realistic compilation by program transformation. In *Symposium on Principles of Programming Languages*, pages 281–292, 1989.

[16] D. Kranz, R. A. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In *Symposium on Compiler Construction*, pages 219–233, June 1986.

[17] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *Transactions on Programming Languages and Systems*, 1(1):121–141, 1979.

[18] MLton, a whole program optimizing compiler for Standard ML. `http://www.sourcelight.com/MLton/`.

[19] S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, 1997.

[20] J. E. Piazza. System for conversion of loop functions in continuation-passing style. US Patent 5881291, Mar. 1999.

[21] J. Reppy. Local CPS conversion in a direct-style compiler. In *Workshop on Continuations*, pages 1–5, Jan. 2001.

[22] D. Tarditi, J. G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *Conference on Programming Language Design and Implementation*, pages 181–192, 1996.

[23] M. Tofte, L. Birkedal, M. Elsman, N. Hallenberg, T. H. Olesen, and P. S. Bertelsen. Programming with regions in the ML Kit (for version 3). Technical Report 98/25, University of Copenhagen, 1998.

[24] A. P. Tolmach and D. Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

[25] K. Yi and S. Ryu. A cost-effective estimation of uncaught exceptions in Standard ML programs. *Theoretical Computer Science*, 237(1), 2000.