Alternate Control-Flow Analyses for Defunctionalization in the MLton Compiler

Author - Ryan Shea (rps3099@rit.edu)
Advisor - Dr. Matthew Fluet (mtf@cs.rit.edu)

May 2016

Abstract

In the compilation of a program, the compiler will translate the written code into a binary capable of being run by a computer. During this process it is common for there to be several intermediate representations of the code to make different parts of the compilation easier, quicker, or more efficient. One of the translations into an intermediate state for functional programming languages is known as closure conversion, which is handled by the defunctionalization pass in the MLton compiler. This process involves the conversion of the higher-order program representation into a first-order program representation through the use of closure applications at every function call site. In this process, Control-Flow analyses (CFAs) interpret the possible functions that can be applied at a now first-order location in the program so that a proper closure of functions that represent the higher-order equivalent at that location can be created. In this paper we look at the precision of a Simply-Typed analysis, 0-CFA, and m-CFA by comparing their overall member size distributions for each of these closure conversions. We use the MLton compiler for Standard ML as the framework within which to analyze these three CFAs.

1 Introduction

In functional programming compilers, closure conversion is the process that transforms the program representation from a higher-order program to a first-order program. One method of accomplishing closure conversion is defunctionalization. This method works by replacing all higher-order functions with a first-order apply function that uses closure sets to accomplish the same functionality as the previously higher-order functions. In this transformation, Control-Flow analysis (CFA) is the process that identifies which first-order function calls could exist at various higher-order function locations. These function calls are then grouped together by the analysis into a collection of constructors known as a flow set that can be used to replace the higher-order function call during the

closure conversion transformation. In order to reduce the cost of performing this analysis, most CFAs overestimate the amount of functions that should be included in any specific closure. This overestimation ensures that no matter what situation is encountered, the first-order program will produce the same output as the higher-order program. [2]

In recent years many new developments have been introduced into the field with the aim to optimize the control-flow analysis of a program such as those found in [4], [5], [6], and [7]. Many of these methods allow for smaller closure estimations that generate faster binaries, while also only having a small cost on the runtime of the process. In this paper we look at three separate CFAs and compare them based on their precision when creating a flow set for each call site. The three CFAs chosen for this paper are a Simply-Typed analysis that uses type information instead of control-flow; 0-CFA which is used currently by production MLton; and m-CFA which is a more precise analysis than 0-CFA. The Simply-Typed analysis represents a proof of concept that the MLton defunctionalization pass can utilize a CFA other than 0-CFA and m-CFA aims to improve the overall precision of the analysis. The main novelty of this paper is its ability to contrast these differing analyses within a uniform production system, the MLton compiler for Standard ML (SML). MLton provides a good framework for this process because its defunctionalization pass, which accomplishes closure conversion for the compiler, is done early on in the process before most optimizations have taken place. This setup allows for a direct comparison and quantification of each analysis' impact on the distribution sizes for the created flow sets, which can then be used to compare the level of precision for each analysis. The expected results for this comparison is that m-CFA will always generate flow sets that are smaller than or equal to the size of those produced by 0-CFA. 0-CFA will in turn have flow sets that are smaller than or equal to those produced by the Simply-Typed analysis. Furthermore it is expected that m-CFA will offer a respectably higher degree of precision than 0-CFA and that 0-CFA will offer a much higher degree of precision than the Simply-Typed analysis.

This paper is organized as follows. Section II covers background information useful for understanding the setting of these comparisons. Section III covers the algorithmic methodology behind each of the three analyses as well as their implementation in MLton. Section IV covers the results of each implementation and discusses what those results likely indicate the precision of each analysis is. Section V covers related work that has been done in the field recently that attempts to increase the precision of control-flow analysis. Section VI concludes this paper's findings. Section VII discusses likely future work in the MLton compiler based on the findings of the paper. Section VIII lists the references used for the project and this paper.

2 Background

In order to understand how defunctionalization and control-flow analysis affects the output binary and compilation, it is important to understand how func-

tional programming languages and the MLton compiler work. In particular the structure of functions in functional programming languages is significantly different than those in an object-oriented paradigm. Lastly it is also important to understand the direct role that control-flow analysis plays in the defunctionalization process in order to understand the important differences between the three analyses presented in this paper.

2.1 Functional Programming

Functional programming is a programming paradigm that is based upon lambda calculus and the mathematical evaluation of functions. This is significant because in pure functional programming the output value of a function depends only on the arguments to that function and has no reliance on the state. Functional programs lack many common features of imperative programs such as loops or other iterative structures, mutable data structures, and saved state when acting in a pure environment. In practical functional programming however, true purity is often unnecessary and thus many functional programming languages, including Standard ML, allow some mutability or state changes into their language for programmer convenience. Additionally while there are no iterative structures in functional programming most languages use pattern matching and recursion in order to produce a comparable program to an iterative loop, which adds overall complexity to the control-flow analysis's job of determining what functions can be called at each call site.

While functional programming varies from imperative programming in a multitude of ways, the most important differences in relation to control-flow analysis and defunctionalization are those that relate to functions themselves. In functional programming functions can be passed as arguments to another function and returned as results from another function. In addition functions themselves are not a simple pointer to a subroutine but are instead a closure representing the function code paired with the value of its free variables. This means simply that functions in functional programming are capable of retaining access to their original scope for the lifetime of the function. All of these features are important characteristics to how functional programming operates, but they make it difficult to translate a functional program into a first-order intermediate representation. In an ideal world the translation would make a perfectly identical copy of the program with a one-to-one relationship between the functions in the original language and the functions in the intermediate language. Unfortunately the types of features unique to functional programming do not exist in the intermediate representation of the program and therefore a best approximation must be substituted. Higher-order functions in particular are difficult to handle as they are functions that are capable of taking other functions as arguments or returning another function after execution. This relationship does not exist inside the binary output by MLton and is the main focus of the defunctionalization pass. It is fairly obvious why this relationship makes it difficult to simply replace a functions call site with a first-order replica of that function. Situations where the arguments to a function are in fact

functions themselves could have variable result types that will now need to be calculated before the application site is capable of executing. In addition in situations where a function's result type was another function, the closure for that function may contain very specific values that were in scope that do not exist at the returned functions execution sites. This is a problem caused by all functions in functional programming being a closure of the function code paired with its free variables. The need to replace a call sites while also referencing values that are in scope only internally in the returned function makes the defunctionalization pass extremely complex. The below example is a good indication of how even using MLton's current 0-CFA analysis to analyze a higher-order program can lead to less than precise results in the defunctionalization pass. As shown in the first-order representation of the program, the result of the fid function remains ambiguous when creating first-order representations for a1 and a2. This leads to the utilization of multi-line case statements to ensure the type safety of those functions instead of being able to easily evaluate the value like a3 is capable of doing. This type of imprecision can be addressed using context aware information to evaluate the possible values of the fid function at each of those variables. This will be demonstrated later when we look at how m-CFA generates control-flow information.

Higher-order program:

```
val fid = fn (f: int -> int) => f
val inc = fn (x: int) => x + 1
val dec = fn (y: int) => y - 1
val neg = fn (z: int) => -1 * z

val a1 = fid inc    (* a1 = inc *)
val b1 = a1 1       (* b1 = 2   *)
val a2 = fid dec    (* a2 = dec *)
val b2 = a2 1       (* b2 = 0   *)
val a3 = neg        (* a3 = neg *)
val b3 = a3 1       (* b3 = -1  *)
```

Flow Analysis:

```
F(fid) = {λ_f}
F(f) = {λ_x, λ_y}
F(inc) = {λ_x}
F(dec) = {λ_y}
F(neg) = {λ_z}
F(a1) = {λ_x, λ_y}
F(a2) = {λ_x, λ_y}
F(a3) = {λ_z}
```

First-order program:

```
datatype t1 = CF1 of unit (*λ_f*)
datatype t2 = CX1 of unit (*λ_x*)
datatype t3 = CY1 of unit (*λ_y*)
datatype t4 = CZ1 of unit (*λ_z*)
datatype t5 = CX2 of unit (*λ_x in F(f) *)
            | CY2 of unit (*λ_y in F(f) *)

fun F ((), f) = f
fun X ((), x) = x + 1
fun Y ((), y) = y - 1
fun Z ((), z) = z * -1

val fid = CF1 ()
val inc = CX1 ()
val dec = CY1 ()
val neg = CZ1 ()

val a1 = case fid of
    CF1 r => F (r, case inc of
                     CX1 r' => CX2 r')
val b1 = case a1 of
          CX2 r => X (r, 1)
        | CY2 r => Y (r, 1)
val a2 = case fid of
    CF1 r => F (r, case dec of
                     CY1 r' => CY2 r')
val b2 = case a2 of
          CX2 r => X (r, 1)
        | CY2 r => Y (r, 1)
val a3 = neg
val b3 = case a3 of
          CZ1 r => Z (r, 1)
```

Figure 1: Example of the defunctionalization of a program using 0-CFA.

In addition to containing higher-order functions, functional programming also contains many cases of currying. Currying is where an n-parameter function can be broken down into a series of 1-parameter functions. Each of those are created through a function taking in a single argument, using that argument in the creation of a new function, and then returning that new function which will accept the next argument. These functions are important to functional programming languages as many different types of programs make use of currying and the key feature that each step of the currying saves the free variables in scope in its closure. A good example of this type of program would be one that unlocks a security lock in a system. The function could be written as taking in two curried parameters and comparing the two for equality to determine if they should unlock the feature or not. The first parameter, the true key value, can then be curried into the function and the resulting function can be used to test for attempts to unlock the feature. This type of curried application provides a lot of value in functional programming as it's a way to save state in a mostly non-mutable environment. The issue for defunctionalization obviously becomes replacing these functions with equivalent values, even though values like the key used to test for equality are no longer in scope of the current environment. These three characteristics as well as many other key features in functional programming make it difficult to directly translate a higher-order functional language into a first order language. The need to ensure as close a match as possible while not taking an unnecessary amount of compilation time has led to the development of a number of CFAs.

2.2 The MLton Compiler

As stated previously MLton is a compiler for the Standard ML functional programming language. MLton has a few unique aspects that impact the defunctionalization process and the Control-flow analysis thereof. The key features of MLton in regards to defunctionalization are its whole-program compilation; its multiple intermediate language representations; and its placement of defunctionalization in the compilation process. These three features together make MLton unique compared to most other compilers, but also serve to make it ideal for a comparison of different Control-flow analyses based on their overall precision. [3]

Compilers in general can be thought of as a program that translates a file from a source language into a machine readable language. This transformation seems relatively straightforward but in actuality offers a lot of places to make different strategic choices. These choices in turn can impact future stages of the compilation and have a cascading affect that changes the entire output of the program. In this way, even two compilers that both compile the same programming language may produce significantly different binaries from the same input source files. The first feature mentioned and one of the main features of MLton is its whole program compilation. Whole program compilation is when a compiler always recompiles a program from scratch. Even if the majority of files are unaffected by the new changes introduced to the program, MLton does

not reuse their machine language formats but recreates them along with the changed files. Though this may seem negative, whole program compilation allows for the compiler to use maximal information when compiling a program, which can greatly improve the performance of the output binary while also reducing its total size. Additionally whole program compilers are much easier to introduce new features into as the compiler itself is simpler due to always having access to the full source program during compilation. The importance of whole-program compilation to this paper is that it means the defunctionalization pass affects the entire program each time it is run. CFA is a whole program analysis but even if a more optimal flow set distribution was found using the new information, a partial program compiler would be unable to utilize that in parts of the program that were not being recompiled. Whole program compilation allows the results of this paper to show a much clearer difference between the three analyses than utilizing a compiler that only recompiles smaller sub-pieces of the program at any one time as the changes in precision can be compared on a larger scale.

In addition to whole-program compilation, MLton also features the use of several intermediate representation languages during compilation. This is significant because any changes from SML to an intermediate language that occur before defunctionalization will impact the program that is analyzed by the CFA. In MLton, an SML program is transformed into an intermediate language known as SXML before the defunctionalization pass takes place. SXML is produced from SML through a defunctorization pass. While defunctorization is outside of the scope of this paper, it does have several key features that are important to understand for the defunctionalization pass. The goals of defunctorization are to turn SML into a polymorphic representation, while exposing the types hidden by functors and signatures, and exposing function calls across modules. One of the key features of this transformation is that all variables bound in **val** and **fun** declarations are renamed while additionally moving all local declarations to the top level. These changes result in the environment keeping track of all variables in scope, which can be utilized by the CFA for context aware analysis of each function's call site. The only other pass after defunctorization and before defunctionalization is a monomorphisation pass. This pass eliminates polymorphism from the language and turns SXML into a simply-typed intermediate language so that the rest of the optimizations have good data representations for all variables and function call sites in the program. This is important for the evaluation of a call site as knowing the type of arguments to and value from a function ensure type safety during the defunctionalization transformation.

The last key feature of MLton, in relation to closure conversion, is the order of execution for the optimization passes. Most SML program compilers start by turning the source program into a higher-order representation first and then running optimizations on that higher-order representation. Only after all the optimizations are done is closure conversion executed to turn the program into a first-order representation of the program, and this transformation is rarely done using defunctionalization. In contrast MLton has a defunctionalization pass that transforms the program into a first-order intermediate language before almost

any other program optimizations are run. The result is that MLton is capable of providing all other optimization passes control-flow information without them having to compute this information themselves. This is also important to this paper because it means that the defunctionalization pass, and as a result the Control-flow analysis of the program, only happens at one point in the compiler. This makes it very easy to compare the results of the three analyses, unlike in other compilers that may have various isolated locations where control-flow information is calculated. This also means that all three analyses are working on an almost untouched representation of the source program, which allows there to be a much greater differential in the results of the three analyses than there may be in a program that has gone through all of its optimization passes before the defunctionalization stage.

2.3 Control-Flow Analysis

Control-flow analysis is the technique used by functional program compilers to solve the higher-order control-flow problem. The higher-order control-flow problem is caused by the inability to guarantee the precise target of a function call in a higher-order language. As shown above the function calls for a1 and a2 were not guaranteed even though we as humans could see that they would eventually evaluate to inc and dec respectfully. In MLton as in many other compilers the control-flow problem is abstracted to a more general value-flow problem. That is to say MLton answers not just the question of what is the target of a function call but instead to what values may an expression evaluate. In the example above a1, a2, and a3 would all have various values associated with them depending upon a variety of factors that a control-flow analysis can consider; such as the scope at their definition, the environment of the program, and even the calling context that they are executed in. CFAs in practice are all algorithms to solve the value-flow problem. A perfect precision solution to the value-flow problem is impossible to compute so CFAs only approximate the solution in such a way that they over-approximate the values that may appear at any one location. If a CFA puts a function call in a flow set then it is not saying that function will be called at that call site but rather that function may be called at that call site. This is an important fact when building a CFA as this over-approximation of the value-flow problem ensures type correctness in the transformation of the higher-order representation into a first-order representation.

The reason the over-approximation of the value-flow problem is important to this paper is because the precision of each closure set is determined based on the best preforming CFA. Each CFA can be measured by how well it is able to approximate the number of members of each of the closures for each call site. The best performing CFA is used as the baseline measurement because as mentioned previously it may be impossible to determine with certainty what the true closure set should be at each call site. In MLton the value-flow problem is solved in such a way that each call site has a specific variable name assigned to it that then points to that call site's closure set. Due to the nature of MLton's optimization orderings there will always be an equal number of flow

sets generated by each CFA for a given program. This makes it possible to compare all three analyses in this paper directly because the overall number of closures created by each analysis will be the same, with only the distribution of closure sizes varying. Although it is outside the scope of this paper it is possible to imagine further comparing the analyses based upon their compilation time and based upon their binary outputs size and runtime. Even though a CFA may generate a higher precision set of closures for the function, it does not guarantee that the increased execution complexity is worth the benefits produced by the more advanced algorithm.

3 Concepts and Implementations

The main contribution of this paper is the implementation and evaluation of three separate control-flow analyses in a production compiler. All three analyses were capable of taking in a real world SML program and producing a valid control-flow for the defunctionalization to utilize. Unfortunately due to the nature of MLton's type safety requirements, it was not possible to fully test the compilation time and output binary of all of these algorithms. This is a result of anything more precise than 0-CFA being treated as unsafe due to the way MLton's defunctionalization ensures type safety for the overall program evaluation. As stated previously, 0-CFA was already implemented and utilized by production MLton and has been not modified for this paper. Simply-Typed analysis was implemented first as a proof of concept that other CFAs could be implemented in MLton without drastically altering the defunctionalization pass. Simply-Typed Analysis also serves as a base case for control-flow analysis as it is one of the simplest methods of answering the value-flow problem as explained in the background section. M-CFA was implemented last to test that a more precise algorithm could be implemented in MLton's defunctionalization pass. The implementation of m-CFA was chosen over other higher precision analyses due to the similarity it shares with 0-CFA and the ease of engineering the algorithm in the MLton compiler's framework.

3.1 Simply-Typed Analysis

Simply-Typed analysis as stated is a very simplistic approach to solving the value-flow problem. In comparison to most CFAs it does not generate a solution based on the interpreted control-flow of the program, but rather based on the type of each function. Since each function is known to have an argument and a return of specific types, it is possible to generate a type based solution to what values can possibly be found at each call site. In order to do so, simply generate a closure for each function by grouping it with all other functions which share its argument and return types. This then leads to the solution that each call site can be properly approximated by the list of all functions that match the input and expected output types of that call site. This solution is obviously correct as all possible functions that could match that call site in the entire program

are present. In addition it serves as a good base case for control-flow analysis because it represents the simplest case of solving the value-flow problem that does any amount of work. While it would technically be a valid solution to say that a call site could be a function from the list of all possible functions, it would not make sense to include in that list functions that are known to not match the argument and return types of the call site.

While Simply-Typed analysis of a program does lead to a valid solution to the value-flow problem, it does so at the cost of all precision. It is easy to imagine the issue with a simple example. If you have a program that defines several small helper functions all taking two integers as inputs and returning an integer as output these functions will quickly explode the closure size of each call site where these functions are found. This can be compounded further by the fact that in a whole-program compiler like MLton, basic math operations will also generate functions of that specific type and will grow both the closure size and the number of call sites where that closure is applied. While it is easy to see that such a solution is bad for generating control-flow information that future optimizations use due to the increased cost of searching through a closure, it can also be bad for the output binary's size and efficiency. In MLton, optimizations such as in-lining occur where small functions can be efficiently eliminated by just duplicating the body of the function at each call site without requiring the overhead of passing arguments into and returning from the function. Using the Simply-Typed analysis to generate function flow sets that are then evaluated for in-lining can lead to poor results as the larger flow sets and uncertainty will reduce the performance of the optimizations pass. These poor results in turn may cause unnecessary growth in the size of the binary and will also affect the runtime negatively due to the increased time spent looking through the larger flow sets. The example below shows how the Simply-Typed analysis's defunctionalization of the earlier example would look. You can see clearly the added ambiguity and increased code sized caused by the lack of precision.

Higher-order program:

```
val fid = fn (f: int -> int) => f
val inc = fn (x: int) => x + 1
val dec = fn (y: int) => y - 1
val neg = fn (z: int) => -1 * z

val a1 = fid inc    (* a1 = inc *)
val b1 = a1 1       (* b1 = 2   *)
val a2 = fid dec    (* a2 = dec *)
val b2 = a2 1       (* b2 = 0   *)
val a3 = neg        (* a3 = neg *)
val b3 = a3 1       (* b3 = -1  *)
```

Flow Analysis:

```
F(fid) = {λ_f}
F(f) = {λ_x, λ_y, λ_z}
F(inc) = {λ_x, λ_y, λ_z}
F(dec) = {λ_x, λ_y, λ_z}
F(neg) = {λ_x, λ_y, λ_z}
F(a1) = {λ_x, λ_y, λ_z}
F(a2) = {λ_x, λ_y, λ_z}
F(a3) = {λ_x, λ_y, λ_z}
```

First-order program:

```
datatype t1 = CF1 of unit (*λ_f*)
datatype t2 = CX1 of unit (*λ_x in F(f), … *)
            | CY1 of unit (*λ_y in F(f), … *)
            | CZ1 of unit (*λ_z in F(f), … *)

fun F ((), f) = f
fun X ((), x) = x + 1
fun Y ((), y) = y - 1
fun Z ((), z) = z * -1

val fid = CF1 ()
val inc = CX1 ()
val dec = CY1 ()
val neg = CZ1 ()

val a1 = case fid of
    CF1 r => F (r, inc)
val b1 = case a1 of
          CX1 r => X (r, 1)
        | CY1 r => Y (r, 1)
        | CZ1 r => Z (r, 1)
val a2 = case fid of
    CF1 r => F (r, dec)
val b2 = case a2 of
          CX1 r => X (r, 1)
        | CY1 r => Y (r, 1)
        | CZ1 r => Z (r, 1)
val a3 = neg
val b3 = case a3 of
          CX1 r => X (r, 1)
        | CY1 r => Y (r, 1)
        | CZ1 r => Z (r, 1)
```

Figure 2: Example of the defunctionalization of a program using the Simply-Typed analysis.

Simply-Typed analysis in MLton was implemented using the defunctionalization architecture by tweaking the algorithm to produce an output similar to that produced by production MLton's 0-CFA. Like 0-CFA, the Simply-Typed analysis walks over the program one time in order to generate the output control-flow. The output itself is represented as a lookup environment structure that maps a variable to the value of its type. In the Simply-Typed analysis, the major change to the solution produced by 0-CFA is that each variable with a function type is mapped to a closure set that contains all functions in the program of that type. This solution is also simpler than the 0-CFA case because it does not require a deeper investigation of the control-flow in the program since only the variables' types matter. This allows for a much more uniform treatment of expressions by the algorithm; although this is unlikely to have a large impact on the runtime of the algorithm as it is still required to walk the entire program to ensure that it covers all variable definition points. The Simply-Typed analysis in MLton does additionally incorporate several features that are not directly related to the analysis itself. While this would likely impact the performance time of the algorithm it should not impact the overall performance of the defunctionalization pass. The features themselves are requirements for the current MLton defunctionalization pass to operate and doing them during the

program walkthrough by the CFA is a convenience that likely saves on overall compilation time by reducing the need to walk over the program separately. It is also important to note that while whole-program compilation is useful for the other CFAs, it has a negative impact on the precision of the Simply-Typed analysis. This is because functions sharing types across modules are grouped into the same closure set when they would otherwise be separated by the piecemeal compilation of a program.

The Simply-Typed analysis overall represents a base case algorithm for solving the value-flow problem while also being a proof of concept that MLton's defunctionalization pass can support additional CFAs without major refactoring. Simply-Typed analysis's implementation showed that as long as a new CFA was able to adapt to the various structures used by 0-CFA and the defunctionalization pass, it was possible to alter the analysis without invalidating the results. The success of the Simply-Typed analysis led to the continuation of this project and the development of the m-CFA algorithm inside MLton.

3.2 0-CFA

0-CFA as stated is the current analysis used by production MLton's defunctionalization pass. The analysis uses a context insensitive approach to solving the value-flow problem, which is where it gets the zero in its name from. 0-CFA functions by abstracting values to the syntax from which they came. In MLton this means that a variable of function type is abstracted to the lambda term that created it and the environment associated with its closure is ignored entirely. This lack of environmental knowledge causes a decent amount of imprecision in the analysis, but leads to a polynomial-time bound, cubic in the worst case, for all variables in the program. The performance of 0-CFA in practice almost always performs better than cubic time and is the reason that MLton originally chose 0-CFA over other more precise analyses, which were computationally more expensive. [1]

0-CFA as an algorithm works by utilizing control-flow information to perform an analysis on the entire program. It does so by creating a flows-to relation for all variables in the program. This information can then be used to generate the possible flow set members that are inserted at each call site. 0-CFA in the purest form can be thought of as following four basic rules to determine flow information for a program. First each lambda term in the program flows to the value representing itself. Second for each function application with a lambda term flowing to the function position and some other value flowing to the argument expression, that other value now flows to the value of the lambda term's argument. Third for each function application with a lambda term flowing to the function position and some other value flowing to the body of the lambda, that other value now flows to the function application's result value. Last any variable flowing to a first-order value simply continues to flow to that first-order value. This is obviously a much more complicated algorithm than the one used by the Simply-Typed analysis but as shown in the example below the results are much more precise because of the extra complexity. The

example from the background section, shown below for convenience, is a good demonstration of the 0-CFA algorithm. The algorithm is capable of analyzing almost all of the values in the program correctly, but cannot quite figure out the proper values for a1 and a2 due to a lack of context that would enable it to determine the value of f at each variables' definition point.

Higher-order program:

```
val fid = fn (f: int -> int) => f
val inc = fn (x: int) => x + 1
val dec = fn (y: int) => y - 1
val neg = fn (z: int) => -1 * z

val a1 = fid inc     (* a1 = inc *)
val b1 = a1 1        (* b1 = 2   *)
val a2 = fid dec     (* a2 = dec *)
val b2 = a2 1        (* b2 = 0   *)
val a3 = neg         (* a3 = neg *)
val b3 = a3 1        (* b3 = -1  *)
```

Flow Analysis:

```
F(fid) = {λ_f}
F(f) = {λ_x, λ_y}
F(inc) = {λ_x}
F(dec) = {λ_y}
F(neg) = {λ_z}
F(a1) = {λ_x, λ_y}
F(a2) = {λ_x, λ_y}
F(a3) = {λ_z}
```

First-order program:

```
datatype t1 = CF1 of unit (*λ_f*)
datatype t2 = CX1 of unit (*λ_x*)
datatype t3 = CY1 of unit (*λ_y*)
datatype t4 = CZ1 of unit (*λ_z*)
datatype t5 = CX2 of unit (*λ_x in F(f) *)
            | CY2 of unit (*λ_y in F(f) *)

fun F ((), f) = f
fun X ((), x) = x + 1
fun Y ((), y) = y - 1
fun Z ((), z) = z * -1

val fid = CF1 ()
val inc = CX1 ()
val dec = CY1 ()
val neg = CZ1 ()

val a1 = case fid of
    CF1 r => F (r, case inc of
                     CX1 r' => CX2 r')
val b1 = case a1 of
          CX2 r => X (r, 1)
        | CY2 r => Y (r, 1)
val a2 = case fid of
    CF1 r => F (r, case dec of
                     CY1 r' => CY2 r')
val b2 = case a2 of
          CX2 r => X (r, 1)
        | CY2 r => Y (r, 1)
val a3 = neg
val b3 = case a3 of
          CZ1 r => Z (r, 1)
```

Figure 3: Example of the defunctionalization of a program using 0-CFA.

0-CFA in MLton was previously implemented as the analysis used by production MLton's defunctionalization pass [1]. 0-CFA as stated earlier walks over the entire program one time in order to generate the output control-flow. The output is represented as a lookup environment structure that maps a variable to the value of its type but does not have any type of context information associated with that variable. 0-CFA has a couple of features that make executing the algorithm as described above easier and more type safe. 0-CFA starts by ensuring that all datatype constructors have their expected arguments represented as distinct type values. 0-CFA then walks over the body of the program using a set of mutually recursive functions. Expressions are broken down by the first function so that each declaration inside the expression is evaluated and visited by the program. The second function checks those declarations for any possible recursive functions and binds them appropriately. It also passes any lambda terms it finds in those declarations to the fourth function during this check.

Any other terms found in the declarations are handed off to the third function. The third function breaks down the expression into a variable, type, and body. These are then used to bind the values found in any function applications as described by the algorithm. In addition the third function hands off any lambdas found to the fourth function while also breaking the body of its input expression into appropriate subexpressions that can be handed back to the first function for further analysis. The fourth and final function is the function that takes a lambda case and binds it to the appropriate value representation of itself. This value is then utilized by the application sites of the lambda as described by the 0-CFA algorithm. The fourth function also uses its evaluation of the lambda's value to hand the body of the lambda back to the first function for further evaluation. This set of mutually recursive functions ensures that every definition point in the program is visited by the analysis and bound to an appropriate value that can be utilized by the defunctionalization pass. This analysis can be done extremely quickly because it analyzes everything from its definition point. The lack of environmental knowledge does affect the overall precision of the algorithm however. Multiple values for the same variable cannot be separated based on the call sites they are utilized in as shown in the defunctionalization example. Additionally the analysis will create flow sets for variables that are defined internal to a body that is never utilized by any call site. These issues with precision were the driving force behind attempting to implement a new and more precise analysis in MLton's defunctionalization pass.

3.3 M-CFA

M-CFA is a recently proposed control-flow analysis that aims to improve the performance of k-CFA, for which 0-CFA is the context insensitive version. M-CFA does this through the use of smart environment representations to reduce the computational complexity required to run the analysis. For m-CFA the analysis keeps track of the scope in which a closure was captured in order to perform a context sensitive analysis. In contrast, k-CFA keeps track of the scope for which each free variable captured in the closure was defined. In practice the two analyses have almost identical precision for program evaluation although k-CFA has a few special cases where it is able to generate a more precise answer than m-CFA. Unfortunately k-CFA is much more computationally expensive than m-CFA due to the resource requirements needed to keep track of each closure's free variables' defining scope. This drastic increase in computational resources led us to choose m-CFA over k-CFA for implementation in MLton. [4]

As stated m-CFA uses a context sensitive approach to solve the value-flow problem. The context for each function closure is based upon the last m call sites lead to the function's application site. In practice this means that for m-CFA and 0-CFA there is almost no difference when m is set to zero because the environments of the call sites are ignored like they would be under the context insensitive 0-CFA. The use of function closure environment information by the non-zero versions of the analysis causes a decent amount of precision to be gained over the base case represented by 0-CFA, but does not increase

computational resources as much as keeping track of all free variables in the environment's scope like k-CFA does. In practice the value for m is either set to one or two because any larger values cause an unnecessary increase in the computation costs without generating a worthwhile return in the precision of the algorithm. For this paper we chose to use one for the value of m, as it most clearly demonstrates the difference in precision between a context insensitive and context sensitive analysis.

M-CFA as an algorithm works in much the same way that 0-CFA does. M-CFA also generates its output by creating a flows-to relation for all variables in the program. The major difference between m-CFA and 0-CFA is the context environment that m-CFA is able to use when creating its flows-to relations. If we consider the four rules used by 0-CFA to generate its relations, we can say that m-CFA more or less uses the exact same rule four. The difference comes from the two application rules, both of which would be seen as a call site in the program, and the lambda terms that are found at those call sites. These call sites generate context information that can be used by the algorithm to improve the precision of its solution for the lambda term. An example of this might be that under 0-CFA a single lambda term was used at four different call sites. The values for the flows-to relation in rules two and three would be capable of distinguishing which call site they were in and keeping their values separate. The value for the lambda term in rule one would not be able to make such a distinction however, and as such the lambda term would flow to four different values found at those call sites. This problem is what m-CFA aims to address by adding in a context aware evaluation of call sites. Since each call site is distinct they can generate information that is distinct for the values at that call site. The lambda term can then use that information to generate a flow to relation for each value that is kept separated by the information. In this way the above example would keep the single distinct value for each of the four call sites evaluations and would add four distinct values for each of the lambda terms evaluations instead of one value representing all four types. These flows-to relations are obviously more precise than 0-CFA generates because they are able to keep distinct information separate as demonstrated further by the example below. As shown in the example, a1 and a2 are able to correctly identify the value of f when they are called in b1 and b2's context. Even though the separate values for f are later unified into a context insensitive representation the increased precision of the analysis still leads to an improvement in the representation of the first-order program.

Higher-order program:

```
val fid = fn (f: int -> int) => f
val inc = fn (x: int) => x + 1
val dec = fn (y: int) => y - 1
val neg = fn (z: int) => -1 * z

val a1 = fid inc     (* a1 = inc *)
val b1 = a1 1        (* b1 = 2   *)
val a2 = fid dec     (* a2 = dec *)
val b2 = a2 1        (* b2 = 0   *)
val a3 = neg         (* a3 = neg *)
val b3 = a3 1        (* b3 = -1  *)
```

Flow Analysis:

```
F(fid) = {λ_f}
F(f) = {λ_x_@b1, λ_y_@b2}
F(inc) = {λ_x}
F(dec) = {λ_y}
F(neg) = {λ_z}
F(a1) = {λ_x}
F(a2) = {λ_y}
F(a3) = {λ_z}
```

First-order program:

```
datatype t1 = CF1 of unit (*λ_f*)
datatype t2 = CX1 of unit (*λ_x*)
datatype t3 = CY1 of unit (*λ_y*)
datatype t4 = CZ1 of unit (*λ_z*)
datatype t5 = CX2 of unit (*λ_x in F(f) @ b1 *)
            | CY2 of unit (*λ_y in F(f) @ b2 *)

fun F ((), f) = f
fun X ((), x) = x + 1
fun Y ((), y) = y - 1
fun Z ((), z) = z * -1

val fid = CF1 ()
val inc = CX1 ()
val dec = CY1 ()
val neg = CZ1 ()

val a1 = case(case fid of
    CF1 r => F (r, case inc of
                     CX1 r' => CX2 r')) of
          CX2 r' => CX1 r'
        | CY2 r' => raise Impossible
val b1 = case a1 of
          CX2 r => X (r, 1)
val a2 = case fid of
    CF1 r => F (r, case dec of
                     CY1 r' => CY2 r')) of
          CX2 r' => raise Impossible
        | CY2 r' => CY1 r'
val b2 = case a2 of
          CY2 r => Y (r, 1)
val a3 = neg
val b3 = case a3 of
          CZ1 r => Z (r, 1)
```

Figure 4: Example of the defunctionalization of a program using m-CFA.

M-CFA in MLton was implemented by changing the 0-CFA algorithm already present to accept and use context information in its evaluation. M-CFA unlike the other two algorithms walks over the program multiple times to generate the control-flow information. Also different from the other algorithms, the output is represented first as a mapping of variable-context pairs to the value of that variable in that context. This information is then compressed down into the lookup environment structure that maps a variable to the value of its type which is used by the rest of the program. In doing so the algorithm loses some context specific information but still retains a higher level of precision than that of 0-CFA. As stated previously, the example shows this relation when a1 and a2 retain their more accurate bindings in comparison to the 0-CFA analysis even though the two values for f are compressed down into a single flow set. M-CFA starts the same as 0-CFA by ensuring that all datatype constructors have their expected arguments represented as distinct type values. M-CFA then walks over the body of the program once in a fashion similar to 0-CFA. It does so in order to perform the setup requirements of the defunctionalization pass, which as stated before are intermingled with the control-flow analysis phase. Unlike the other two algorithms m-CFA requires free variable information and as such it also runs the free variable generation pass after the initial setup requirements have been fulfilled. M-CFA then walks over the entire program again using a similar

set of mutually recursive functions to those used by 0-CFA but with context information included. Expressions in the program are treated almost identically and are again broken down by the first function so that each declaration inside the expression is evaluated and visited by the program. The second function checks those declarations for any possible recursive functions and binds them in the current context as appropriate, but it does not pass on lambda terms as it did previously. Any other terms found in the declarations are handed off to the third function as before. The third function breaks down the expression into a variable, type, and body once again. Here the first-order values are treated the same as before with the value being set appropriately and the body being broken into subexpressions that can be handed back to the first function for further analysis. Unlike before, lambda terms found are not handed to the fourth function as they are taken care of instead during the application case. The application case is different from the previous implementation because it now hands the lambda term found in its function variable to the fourth function for evaluation, along with a new context representing evaluating at the current variables definition point. The results from this evaluation are then collected and the variables for the application case are bound to the old context with the results from the new context. This step is vital for bridging the flows to relation between variables in different contexts. The fourth and final function is the function that takes a lambda case and binds it to the appropriate value representation of itself in the newly given context. The fourth function again uses its evaluation of the lambda's value to hand the body of the lambda back to the first function for further evaluation. It also now checks to make sure the lambda term has not been evaluated in the current context to avoid an infinite loop situation while evaluating recursive functions. This new set of mutually recursive functions ensures that every call site in the function generates appropriate context information that can be used by the algorithm to perform a more precise analysis of the program than done by 0-CFA. This analysis is not as quick as 0-CFA because it walks over the program multiple times and can loop multiple times on function application points, but in practice these should not be serious issues that drastically raise the execution time of the algorithm. Furthermore the use of context information to generate a more precise answer even has the possibility of improving MLton's overall compilation time by reducing the information sent to future optimizations, thus increasing their performance.

M-CFA overall represents a more precise algorithm for solving the value-flow problem than 0-CFA. M-CFA's implementation demonstrated that the defunctionalization pass could have a more precise algorithm implemented in the current framework. As previously stated the type-checking done by MLton to ensure type safety does not allow m-CFA to be fully compiled, however the hope is that future work in MLton will be capable of completely using the analysis to get a full comparison between m-CFA and 0-CFA.

4 Results and Discussion

The results for this paper were collected by comparing the distribution of flow set sizes produced by each analysis. Each analysis was modified to output all of the flow sets it generated during the analysis phase of defunctionalization. The size of those flow sets were then collected to determine the distribution produced by that analysis. These results were then saved for comparison with other analyses and test runs. In order to do a total comparison of the results between these algorithms, all three analyses were run first on three test functions to get a general idea of how they performed. The files chosen for those tests were the three largest benchmark files in the MLton benchmarking suite. The names of the files were hamlet, model-elimination, and mlyacc in order from largest to smallest. The results from each of these were normalized and then compared against the other test runs. This allowed us to investigate if there was a firm trend in how each analysis' distribution of flow sets appeared. After those results were collected and confirmed, the analyses were run on the MLton source code to produce a large real world result for direct comparison. The results for the MLton source code were then normalized as well and compared directly against each other to determine which analysis had the highest precision. The results for this experiment show what was originally predicted when implementing all the various analyses; Simply-Typed has worse precision than 0-CFA, which in turn has worse precision than m-CFA.

As stated, we first compared the Simply-Typed analysis using three trial runs of MLton's benchmark files in order to determine if a trend already existed in the analysis' flow set distribution. As shown in the table and graph below, the Simply-Typed analysis has a definite trend with a decent resemblance to a cumulative probability distribution. The analysis also has a large percentage of flow sets with more than six members, which was relatively large for the benchmark files when comparing against the other two analyses.

Number of Lambdas in Each Flow Set	hamlet	model-elimination	mylacc
0	0.0017	0.0005	0.0016
1	0.3428	0.3954	0.2533
2	0.5526	0.5479	0.3366
3	0.6225	0.5990	0.3775
4	0.6675	0.6750	0.5016
5	0.7043	0.7142	0.5343
6	0.7370	0.8323	0.6127
7+	1.0000	1.0000	1.0000

Table 1: Results for the Simply-Type analysis' three trial runs.

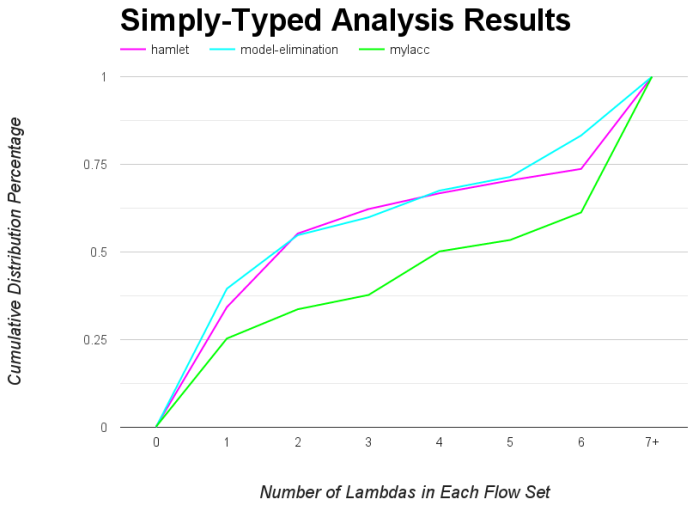

Figure 5: Cumulative distribution graph for the flow sets produced by the Simply-Typed Analysis.

The results for the Simply-Typed analysis' test runs show that the Simply-Typed analysis is definitely not very precise. Compared to the results of 0-CFA and m-CFA shown below, a large number of the Simply-Typed flow sets have a significantly higher number of members. In particular both 0-CFA and m-CFA have well above 90 percent total distribution after just creating flow sets of size one. In comparison the Simply-Typed analysis does not reach 90 percent total distribution in any of the trial runs with flow sets of six members or less. While Simply-Typed acts as a great base case analysis and severed its purpose as a proof of concept, its actual performance when it comes to program analysis is too weak to be utilized by any real world production system.

After running our test trials on the Simply-Typed analysis we moved on to 0-CFA. The results for this were expected to be much better than Simply-Typed's results and proved to be so. Additionally the existing trend of set distributions was much clearer for 0-CFA than it was for the Simply-Typed runs. As the table and graph below show, almost all of the runs shoot up to around 95 percent cumulative distribution with flow sets of size one. After that there is a gradual exponential rise until 100 percent cumulative distribution is reached.

Number of Lambdas in Each Flow Set	hamlet	model-elimination	mylacc
0	0.0082	0.0033	0.0245
1	0.9428	0.9891	0.9673
2	0.9741	0.9976	0.9935
3	0.9871	0.9991	0.9984
4	0.9942	0.9995	1.0000
5	0.9949	1.0000	1.0000
6	0.9952	1.0000	1.0000
7+	1.0000	1.0000	1.0000

Table 2: Results for 0-CFAs' three trial runs.

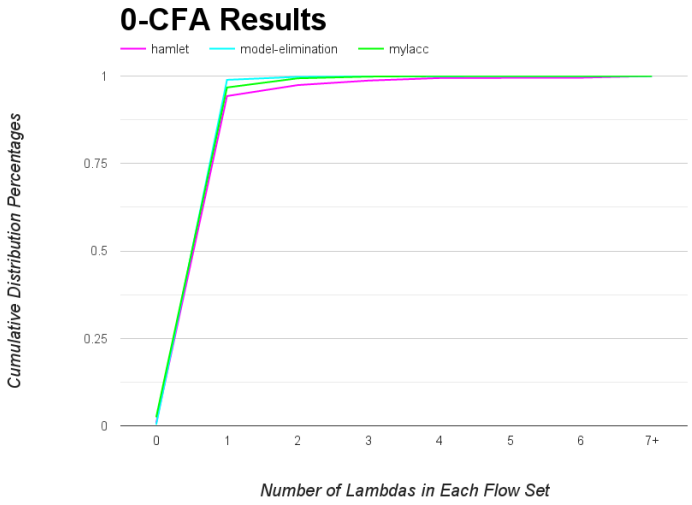

Figure 6: Cumulative distribution graph for the flow sets produced by 0-CFA

The results above show that 0-CFA is a pretty precise analysis. The results are well distributed with only about four percent of flow sets having more than one member on average. Additionally only in one trial run did the number of lambdas in a flow set exceed six members. These results are very encouraging as it shows that 0-CFA is a good analysis despite how fast its algorithm is. The only real bad result the algorithm had is with its zero member flow set distribution. Here the highest distribution out of all three runs was around 2.5 percent and the other two did not even exceed one percent. This result shows that by using definition points instead of function application sites to evaluate lambda terms, your algorithm loses a lot of precision at identifying cases where a defined value is never utilized. Despite this, the results clearly show that 0-CFA is a good analysis and that m-CFA has to perform exceptionally well in order to justify the increased performance cost of using its context sensitive algorithm.

The final test trials we ran evaluated m-CFA on the same three files as the other two analyses. The results show that m-CFA performs extremely well. As the table and graph below show, m-CFA has a similar trend to that of 0-CFA with two caveats. The first caveat is that m-CFA's zero distribution is always at least 50 percent of the total cumulative distribution of flow sets and on average much higher. The second caveat is that m-CFA always hits 100 percent cumulative distribution for the three test runs with flow sets of size two members or less. This is a significant improvement over both of the other analyses but especially demonstrates why the Simply-Typed analysis' weak results are not worth considering for a production environment.

Number of Lambdas in Each Flow Set	hamlet	model-elimination	mylacc
0	0.8845	0.8436	0.5294
1	0.9939	0.9910	0.9706
2	1.0000	1.0000	1.0000

Table 3: Results for m-CFAs' three trial runs.

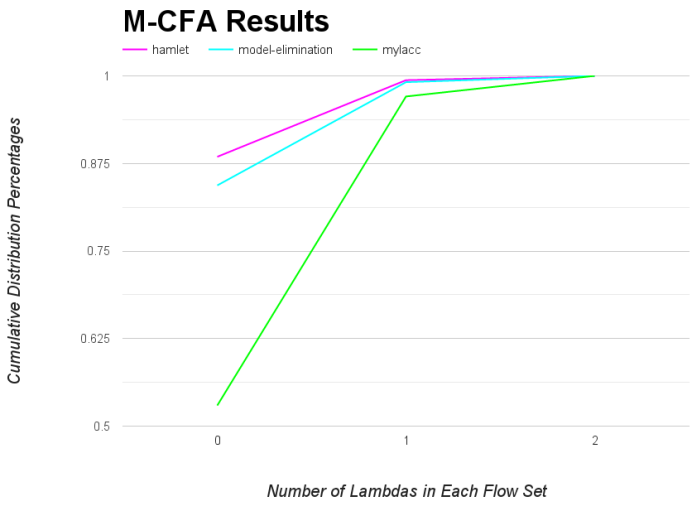

Figure 7: Cumulative distribution graph for the flow sets produced by m-CFA

As stated before 0-CFA had really impressive results that would make it difficult to justify implementing a more computationally expensive control-flow analysis. Despite this the precision of m-CFA, even with the context depth set to one, is significantly better than that of 0-CFA for all of the trial runs. These results show that m-CFA generates not only more zero and one member flow sets, but also is capable of generating a distribution of flow sets that reaches 100 percent cumulative distribution at a very small member size. While we do not have results for how other optimization passes in MLton would use such information, it is possible to imagine that the much smaller average size of flow sets could lead to significant performance gains where the optimization regularly cycles through the values in each flow set.

To verify that the results we collected from the trial runs were accurate on a larger scale we decided to test our three analyses on the MLton source code. The code base for MLton was extremely large compared to the trial runs and provided a total of 39,429 variables that each analysis assigned flow sets to. The overall distribution as shown in the table and graph below was comparable to that seen in the test runs for each algorithm. An interesting result of the comparison between the analyses was that despite performing quite well in the trial runs, 0-CFA still generated a flow set with over a thousand members. In comparison, m-CFA generated only one flow set that was larger than ten members, which happened to contain a relatively small 31 total members.

Number of Lambdas in Each Flow Set	Simply-Typed	0-CFA	M-CFA
0	0.0005	0.0034	0.9071
1	0.2672	0.8861	0.9978
2	0.3522	0.9699	0.9996
3	0.3965	0.9742	0.9996
4	0.4178	0.9788	0.9998
5	0.4370	0.9841	0.9998
6	0.4471	0.9851	0.9998
7	0.4597	0.9859	0.9999
8	0.4689	0.9871	0.9999
9	0.4780	0.9875	0.9999
10 - 99	0.6719	0.9957	1.0000
100 - 999	0.8359	0.9998	1.0000
1000+	1.0000	1.0000	1.0000

Table 4: Results for all three analyses' compilation of MLton source code.

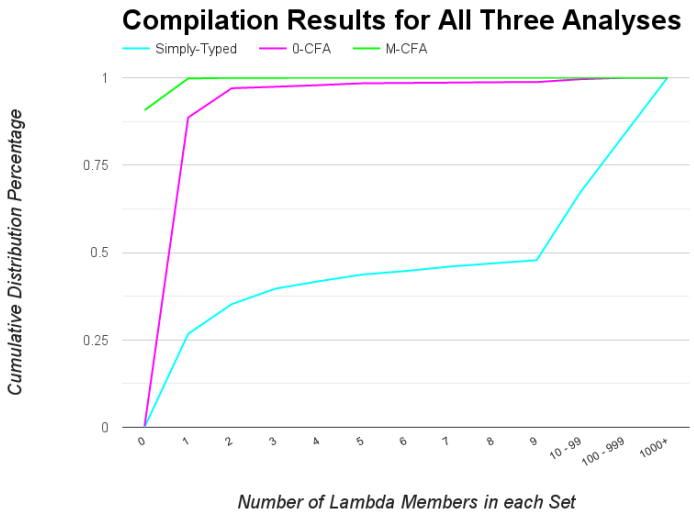

Figure 8: Cumulative distribution graph for all three analyses' compilation of MLton source code.

As the results show, m-CFA performed much better than 0-CFA did when comparing their analyses of the MLton source code. The results above also show again that one of the most impressive ways in which m-CFA outperforms 0-CFA is in the creation of zero member flow sets. While the overall results show that the distribution for flow sets with two or more members is relatively comparable, m-CFA is much better than 0-CFA at generating extremely small flow sets. This extra precision is exactly what this paper was looking to find when implementing the algorithm and also demonstrates that it is possible to improve the defunctionalization pass of MLton significantly by just changing the CFA used to generate control-flow information for the transformation.

5 Related Work

While control-flow analysis has existed for a long time in the field of computing, there has been a recent renaissance that has led to many advancements and new algorithmic proposals. M-CFA, CFA2, and Pushdown CFA are all recent developments that aim to improve the control-flow analysis of compilers through new techniques. This paper decided to implement m-CFA, described in [4], as it is the most similar to production MLton's 0-CFA, but all of the algorithms above have been shown to offer more precision than typical 0-CFA does.

CFA2 is a new algorithm developed in 2010 by the authors of [5] that aimed to keep a context free approach to control-flow analysis, while still improving on the overall precision of the process. CFA2 aims to have precise call and return matching even in the presence of higher-order function calls. Most CFAs approximate the valid control-flow of a program as a set of all paths through a finite graph of abstract machine states. This can lead to issues however as recursive function calls are hard to approximate in such a representation and can lead to difficulties when analyzing higher-order languages. Instead of taking this approach, CFA2 uses a pushdown model of the program where you first compute basic blocks for a procedure and then use those blocks to compute the data-flow at a call-node. In particular CFA2 uses a stack and heap for variable bindings which leads to significant precision gains over the abstract machine states. It also allows for the removal of bindings that are no longer valid, which is not possible in 0-CFA since it does not have any context awareness. These changes overall lead CFA2 to be more precise than 0-CFA while also normally visiting a smaller state space due to visitation of procedures being based on the call-nodes.

Pushdown flow analysis is an advancement that uses the information discovered in the creation of CFA2 and takes it a step farther by incorporating abstract garbage collection into the process. CFA2 made it possible to precisely match the calls and returns of recursion without approximating the stack like k-CFA does. The authors of [6] decided to use this while addressing the other issues faced by that type of pushdown model. The cross-flow problem is an example of a problem where a simple pushdown analysis will still get confused. The cross-flow problem refers to the case where a function called in the same environment with two different call sites will often times merge the resulting closures for those

call sites together, especially when there are unreachable bindings introduced in the environment. In order to solve this issue the authors used techniques from abstract garbage collection. Once a value for a call site has been assigned, Pushdown CFA is able to utilize abstract garbage collection to ensure that there is not an intermingling of the closure value for the yet unanalyzed call site in the same environment. This leads to an increase in the overall precision of the algorithm and an overall increase in the precision of the analysis over CFA2.

While both CFA2 and Pushdown CFA are much more precise than 0-CFA, they do have computational complexity drawbacks. In their worst case CFA2 is $O(2^n)$ and Pushdown CFA is $O(n^6)$. In comparison 0-CFA is worst case $O(n^3)$ but in practice almost always preforms better than that. Additionally while both algorithms are proven to be more precise they are difficult to implement in a production compiler and require a significant amount of customization to enable their analyses. This led the authors of [7] to attempt to develop a better pushdown solution that would lower the computation complexity of the algorithm while additionally making implementation in a real world compiler much easier. Their final solution was Pushdown CFA for Free (PCFAF) which had a constant-factor overhead and cost of $O(n^3)$ in the monovariant case. The authors accomplished this by utilizing methods from the abstracting abstract machines (AAM) approach to constructing static analyses. AAM derives a function for approximating program behavior starting with the abstract machine semantics of a language. This allows AAM to produce a finite state abstraction which is inherently imprecise when it comes to reasoning, as is seen with the k-CFA algorithm, but incredibly efficient. PCFAF overcomes this imprecision by combining the AAM approach with the recent advancements from Pushdown CFA. This is accomplished by having the continuation address for a function be a simple polyvariant entry point, which in turn is represented by its expression and abstract binding environment. By introspecting on entry points and then using its choice of continuation from the AAM approach, PCFAF yield a finite-state analysis whose call transitions are precisely matched with return transitions. Furthermore this matching requires no run-time or development-time overhead making it a significant improvement on standard Pushdown CFA. This algorithm was another candidate for implementation in the MLton compiler and was ultimately passed over simply due to the ease of implementing m-CFA in the MLton defunctionalization framework compared to implementing PCFAF.

6 Conclusion

The motivation for this project was to implement a new control-flow analysis in the MLton compiler's defunctionalization pass that improved the overall precision of the analysis. In order to do this we set out first to implement the Simply-Typed analysis to demonstrate that it was possible to change the analysis used by the defunctionalization pass, without reworking the entire transformation. After verifying that the Simply-Typed analysis worked, we decided to implement m-CFA as an analysis that had higher precision than 0-CFA and

would fit well within the current defunctionalization framework. We were able to successfully implement m-CFA and the results from our trial runs and compilation of the MLton source code show that m-CFA is capable of generating a much more precise distribution of flow sets than 0-CFA. Unfortunately we were unable to further compare the two analyses as MLton is not currently able to handle analyses that are more precise than those 0-CFA generates.

This project overall can be said to have accomplished all of its goals. Simply-Typed analysis was implemented as a proof of concept that we anticipated being much less precise than production MLton's 0-CFA and the results verify that. Additionally we expected there to be a much more dispersed distribution of the flow sets than the other two algorithms generated and the results show that such a distribution does exist in the data. 0-CFA performed as well as we expected and the results show why it is such a popular CFA even without context sensitive analysis. M-CFA was implemented to try and improve the precision of 0-CFA by an amount that would justify fully modifying the MLton defunctionalization pass to support a more precise analysis. The actual results from m-CFA exceeded our expectations and demonstrated a high possibility that the overall compilation of programs in MLton could be improved by adopting the more powerful analysis. In addition the overall hypothesis that the Simply-Typed analysis would generate flow sets equal to or larger than 0-CFA, which would in turn generate flow set equal to or larger than m-CFA held true for all of our comparisons. This shows that a definite precision hierarchy exists between the analyses and further testing on their binary output's size and runtime is needed to finalize the comparisons.

7 Future Work

Our goal with this project was to demonstrate that it was worthwhile to refactor the defunctionalization pass to allow for analyses with higher precision than 0-CFA. Now that we have accomplished this goal, immediate future work will likely be working towards adapting the defunctionalization to handle more precise analyses. One option to attempt this is to adjust the type-checking to use the Simply-Typed analysis' results to ensure type safety. Any analyses that generates a flow set for a variable must have that flow set be a subset of the one generated by the Simply-Typed analysis. This solution requires the caveat that a default case be added to flow sets that error if the correct function is not found in the flow set. Other options include raising the type checking's precision estimate to that of m-CFA, but this will only work for implementing functions that are of lower precision than m-CFA which defeats the point of the change. Besides adjusting type checking to be more forgiving of high precision analyses, the defunctionalization pass in MLton could also be refactored significantly. As stated in this paper there are currently several setup steps being run concurrently with the control-flow analysis phase that may be preventing certain algorithms from being implemented. A future task that would be beneficial to MLton would be to decouple those steps from the analysis phase. Additionally

a separate future task could be to modify several other phases in the defunctionalization pass to be more modular so that future analyses can rearrange them as necessary. All of these tasks combined should they be implemented would allow for a high level of experimentation with implementing alternate analyses.

Once the defunctionalization pass has been refactored to accept more algorithms, the next step for this project will be to compare the results for the three analyses presented here in terms of compilation time and output binary size and runtime. These results will demonstrate conclusively one way or the other whether the increased precision of MLton's defunctionalization pass under m-CFA is worth the increased computational complexity. After those results are collected, future work that may be beneficial is to implement some of the other higher precision analyses, including those presented in the related works section of this paper, in order to determine what the most optimal analysis is for MLton. While this paper used the Simply-Typed analysis as a base case to compare the precision of 0-CFA and m-CFA, future comparisons would most optimally use 0-CFA as a base case. This would demonstrate that all future implementations of new analyses aim to increase the precision of the current production MLton defunctionalization pass in some measurable way.

References

[1] Henry Cejtin et al., Flow-Directed Closure Conversion for Typed Languages, Proceedings of the 9th European Symposium on Programming Languages and Systems, p.56-71, April 02, 2000

[2] Jan Midtgaard. 2012. Control-flow analysis of functional programs. ACM Comput. Surv. 44, 3, Article 10 (June 2012), 33 pages.

[3] MLton. http://mlton.org. Web. May 2016.

[4] Matthew Might, Yannis Smaragdakis, and David Van Horn. 2010. Resolving and exploiting the k-CFA paradox: illuminating functional vs. object-oriented program analysis. SIGPLAN Not. 45, 6 (June 2010), 305-315.

[5] Dimitrios Vardoulakis and Olin Shivers. 2010. CFA2: a context-free approach to control-flow analysis. In Proceedings of the 19th European conference on Programming Languages and Systems (ESOP'10), Andrew D. Gordon (Ed.). Springer-Verlag, Berlin, Heidelberg, 570-589.

[6] J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. Pushdown flow analysis with abstract garbage collection. Journal of Functional Programming, 24(2-3), 2014.

[7] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown control-flow analysis for free. SIGPLAN Not. 51, 1 (January 2016), 691-704.