Cooking with Gas:
Making an Optimizing Curry Compiler

Steven Libby

November 25, 2018

Abstract

It is a common belief in the Curry community that we can pass off the work of compiler optimizations to more established compilers such as GHC and SICSTUS-Prolog. While this seems reasonable, I will show that it has major consequences for the performance of Curry code. Furthermore I show that these performance problems are a direct result of passing optimizations to other compilers. I propose that in order to make Curry programs fast, we need to do the work of optimizing Curry programs ourselves. To this end, I will develop an optimizing Curry compiler to demonstrate that Curry can be optimized, and that it is beneficial to do so.

1 Introduction

Using functional logic programing we can solve difficult problems in a very small amount of code. As an example, we can solve the n-queens problem, in just 5 readable lines of code. We can write simple declarative programs for many complex tasks. Unfortunately, many computer scientists consider current implementations of functional logic languages too slow to use in practice. This has led to the idea that functional logic languages are inherently slow. I believe a more likely culprit is the fact that very little work on program optimization has been incorporated into modern functional logic compilers. In this dissertation I intend to develop an optimization framework for the functional logic language Curry. This will include an intermediate language suitable for optimization; implementations of the deforestation, unboxing, and shortcutting optimizations; justification of the correctness of the implementations; a compiler to a low level language; and analysis of the running time and memory usage on several Curry programs. I will show that it is possible to apply the optimizations of unboxing, deforestation, and shortcutting to Curry programs. Furthermore, these optimizations will improve the performance of Curry programs.

Functional logic programming is a very powerful technique for expressing complicated ideas in a simple form. Curry implements these ideas with a clean, easy to read syntax. It is syntactically very similar to Haskell, a well known

functional programming language. It is also lazy, so evaluation of Curry programs is similar Haskell as well. Curry extends Haskell with two new concepts. First, there are non-deterministic functions, such as "?". Semantically `a ? b` will evaluate `a` and `b` and will return both answers to the user. Second, there are free, or logic, variables. A free variable is a variable that is not in the scope of the current function. The value of a free variable is not defined, but it may be constrained.

Consider the following Curry code for solving n-queens:

```
queens = isEmpty (set1 unsafe p) = p
 where p = permute [1..n]

unsafe qs
 | qs =:= (xs++[a]++ys++[b]++zs) = abs (a-b) =:= length ys
  where xs, a, ys, b, zs free
```

In the `unsafe` function I've broken the list `qs` into 5 pieces. two of the pieces, `a` and `b`, are lists with a single element. the sublists, `xs`, `ys`, and `zs` are free to be as long as they want. However, I've constrained the total list `(xs++[a]++ys++[b]++zs)` to be the same as our argument `qs`. the effect is that `a` and `b` are arbitrary elements in the list `q`, and `ys` is the list of elements between `a` and `b`.

Free variables are given concrete values in Curry programs through narrowing. The semantics of narrowing and non-determinism in Curry are given by Antoy et al. [10]

There are currently two mature Curry compilers, Pakcs and Kics2. Pakcs compiles Curry to Prolog in an effort to leverage Prolog's non-determinism and free variables. Kics2 compiles Curry to Haskell in an effort to leverage Haskell's higher order functions and optimizing compiler. Both compilers have their advantages. Pakcs tends to perform better on non-deterministic expressions with free variables, where Kics2 tends to perform much better on deterministic expressions. Unfortunately neither of these compilers perform well in both circumstances.

Sprite, an experimental compiler, aims to fix these inefficiencies. The strategy is to compile to a virtual assembly language, known as LLVM. So far Sprite has shown promising improvements over both Pakcs and Kics2 in performance, but it is not a mature compiler.

One major disadvantage of all three compilers is that they all attempt to pass off optimization to another compiler. Pakcs attempts to have Prolog optimize the non-deterministic code; Kics2 attempts to use Haskell to optimize deterministic code; and Sprite attempts to use LLVM to optimize the low level code. Unfortunately none of these approaches works very well. While some implementations of Prolog can optimize non-deterministic expressions, they have no concept of higher order functions, so there are many optimizations that cannot be applied. Kics2 is in a similar situation. In order to incorporate non-deterministic computations in Haskell, a significant amount of code must be threaded through each computation. This means that any non-deterministic expression cannot be optimized in Kics2. Finally, since LLVM doesn't know

about either higher order functions or non-determinism, it loses many easy opprotunities for optimization.

Curry programs have one last hope for efficient execution. Recently, many scientists [63, 68] have developed a strong theory of partial evaluation for functional logic programs. While these results are interesting, partial evaluation is not currently automatic in Curry. Guidance is required from the programmer to run the optimization. Furthermore, the optimization fails to optimize several common programs.

So far all of these approaches have failed to include into their implementations the large body of work on program optimizations. This leads to the inescapable conclusion that Curry needs an optimizer. The purpose of this proposal is, first, to make a case that Pakcs and Kics2 both fail to do an adequate job of optimizing Curry programs, then to propose a new solution of building optimization into the Curry compiler pipeline.

This may seem trivial at first. After all, optimization has been studied heavily for decades, and there are many well understood optimizations for imperative, functional, and logic programming. [2,3,5–7,19,20,35,36,40–42,54,60, 64,66,71,73,75] Why don't I just use these optimizations in Curry? Ideally I can, but it's not immediately clear that these optimizations are valid in Curry. This dissertation will contain two components: implementing some optimizations in Curry, and proving the correctness of these optimizations.

I'm limiting myself to implementing deforestation, unboxing, and shortcutting in this dissertation. While there are many other optimizations that I could implement, I believe that these will be the most beneficial.

The rest of this paper is organized as follows. Section 2 presents the Curry language. Section 3 presents the need for an optimizing compiler in Curry. Section 4, 5, and 6 present the history of compiler development for functional, logic, and functional logic languages respectively. Section 7 presents the approaches to compiling Curry specifically. Section 8 presents the optimizations I plan to implement. Section 9 presents my final deliverable at the end of the dissertation. Finally section 10 concludes.

2 Curry Language

There are traditionally two styles of programming that are considered declarative. Functional programming uses functions as an abstraction to encourage higher order reasoning. Logic programming uses relations as an abstraction to encourage relational thinking. Curry is an attempt to unify both of these approaches as a functional logic language.

Curry is by no means the first language to integrate the ideas of functional and logic programming. On the logic side, Mercury [70] and HAL [30] included a type system and higher order functions; Ciao [50], a dialect of Prolog, added notation for functions; OZ [69] also included support for both functional and logic programming; and the Toy programming language [25] adds support for lazy functional logic programming in a Prolog-like syntax. Curry [47] is a modern

functional logic language with Haskell-like syntax.

Curry is syntactically similar to Haskell, however there are a few differences semantically. A Curry program is defined as a collection of function, where each function is given a list of defining equations $lhs = rhs$. To avoid confusion with equations defined later, and because of their relation to rewrite rules, we refer to these equations as rules. A partial specification for the syntax of Curry is given below. While this specification is not complete, it is enough to describe the concepts in this dissertation.

Program	→	$(Data \mid Function)^*$	
Data	→	data $ID = (ID\ Type^*)\ (\mid ID\ Type^*)^*$	
Function	→	$ID :: Type \mid Rule$	
Rule	→	$ID\ Pattern^* = Expr$ [where $Decl^+$]	
	|	$ID\ Pattern^*(\mid Expr = Expr)^*$ [where $Decl^+$]	
Pattern	→	$constructor\ Pattern^* \mid variable \mid literal$	
Expr	→	$Expr\ Expr$	function application
	|	ID	variable or function
	|	$literal$	number, boolean, or character
	|	$Expr$ op $Expr$	binary operation
	|	$(Expr)$	
	|	let $Decl$ in $Expr$	let expression
Decl	→	ID^+free	
	|	$Pattern = Expr$ [where $Decl^+$]	
	|	$ID\ Pattern^* = Expr$ [where $Decl^+$]	
	|	$ID\ Pattern^*(\mid Expr = Expr)^*$ [where $Decl^+$]	

Curry has three semantic categories for names in programs. A *function name* is any name that is defined by *Function*, and a *constructor name* is any name defined by *Data*. All other names are variables. An expression is a *value* if it only contains constructor names or literals. There is a distinguished function called `main` that contains no arguments. A Curry program is executed by evaluating the `main` function.

Those familiar with functional programming may be surprised when learning Curry. The first surprise is that the order of rules does not matter. We can define the following function:

```
f 1 = 1
f n = 2
```

While we may reasonably expect the expression `f 1` to evaluate to `1`, this is not true in Curry. Curry will find every possible evaluation of an expression. Since `f 1` can also match the second rule, the expression is evaluated to both `1` and `2`. This is an example of non-deterministic evaluation. A Curry program will evaluate an expression to every possible value.

There is one important non-deterministic function in Curry. The "`?`" function will non-deterministically select one of its two operands. We define the function as:

```
x ? y = x
x ? y = y
```

This function allows us to construct non-deterministic expressions without defining new functions. It will be used repeatedly throughout the rest of this proposal.

Another surprise in Curry is the addition of *free variables*. A free variable does not have a value, however we can constrain its value.

Consider the function:

```
last xs
 | (ys++[y]) == xs = y
  where y, ys free
```

The variables `ys` and `y` are free, so their value is not determined. However we made the constraint that `ys ++ [y]` is equal to the input. This ensures that `y` is the final element of the list.

A final surprise is that expressions are allowed to fail. An expression fails if it cannot produce a value. For example `head []` will fail, because you cannot take the head of an empty list. If we have a non-deterministic expression `a ? b` and `a` fails, then Curry will only return the result of `b`. We frequently use the behavior to constrain non-deterministic expression by defining equations. Consider the following rule:

```
invert f y
 | f x == y = x
   where x free
```

This `invert` function asks for all solutions `x` such that the equation `f x == y` is satisfied for any unary function `f`.

There are many other extensions to Curry including functional patterns and default rules, however these extensions can be converted to vanilla curry.

2.1 Evaluation

In order for Curry to be a useful programming language, we must be able to run Curry programs. The theory of functional logic programming is rooted firmly in term rewriting [22,55], and graph rewriting [31,67] in particular. Rewriting is a purely syntactic system for transforming expressions based entirely on a set of *rewrite rules*. In Curry, the rewrite rules correspond to the function rules. For example, the rules for reversing a list, and appending two lists are given below

```
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]

[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

With these rules the expression `reverse [1,2] ++ reverse [3,4]`, rewrites to `(reverse [2] ++ [1]) ++ reverse [3,4]`. Rewriting continues until the expression is in a *normal form*, a form in which no more rewrite rules apply. In Curry, normal forms are values or failed computations.

Formally, a Curry expression is a rooted, labeled, directed graph. Each node has a label and a list of children. If e is an expression, then a subexpression s of e is a subgraph of e such that s has a root, and there is no edge from a node in s to any other node in e. In other words, a subexpression must also be an expression. A *path* from the root of e to the root of a subexpression s is a sequence of integers such that the first integer corresponds to a child c of the root of e, and the rest of the sequence is a path from c to s. We write $e|_p = s$ to say the there is a path, p, from the root of e to the root of s. The notation $e[r]_p$ means: replace the subexpression at position p with expression r, where replacement is defined in the usual way.

A rewrite rule $l \to r$ is a pair of expressions, and s substitution σ is a function from variables to expressions. A rewrite rule $l \to r$ can be applied to an expression e, if there is a subexpression s and substitution σ, such that $\sigma(l) = s$. We write $e \to_{l \to r,p,\sigma} e'$ to mean a rewrite step where we replace the subexpression at p with the $\sigma(r)$. A *narrowing step* is a rewrite step where we unify [55] s and l with substitution σ, then we replace $\sigma(s)$ with $\sigma(r)$. We write $e \leadsto_{l \to r,p,\sigma} e'$ to mean e narrows to e', with rule $l \to r$, at position p, using substitution σ.

This is a lot of notation, so lets apply it to the reverse example above. if we want to apply a narrowing step to the expression `reverse [1,2] ++ reverse [3,4]`, we need to find a subexpression that will unify with a rule. One subexpression we can pick is `reverse [1,2]` to unify with `reverse (x:xs)`. This subexpression is at position 1 of the expression, since it is on the left hand side of `++`. Our substitution is σ = {`x` $\to$ `1, xs` $\to$ `[2]`}. Finally, our rewrite step is

`reverse [1,2] ++ reverse [3,4]` $\leadsto_{R,p,\sigma}$
`(reverse [2] ++ [1]) ++ reverse [3,4]`
where
R = `reverse (x:xs)` $\to$ `reverse xs ++ [x]`,
p = `[1]`, and σ = {`x` $\to$ `1, xs` $\to$ `[2]`}.

Now with this terminology, evaluating a Curry expression is actually very simple. First, find a rewrite rule where the left hand side will unify with a subexpression, then narrow the expression using that rule. Repeat this process until a normal form is found.

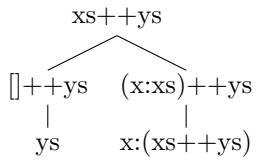
The astute reader may have noticed that I didn't say anything about how we pick the rule, path, or substitution. The process of selecting rules is called the *rewriting strategy*. We use a needed narrowing rewriting strategy [14]. We omit most of the details, because they're long, complicated, and not relevant to an optimizing compiler.

There are two relevant ideas from the needed narrowing strategy. The first is that needed narrowing is a lazy narrowing strategy. Curry programs follow the constructor discipline. The symbols are partitioned into two sets: functions

symbols which have an associated rewrite rule, and constructor symbols which do not. An expression is in head normal form if it is rooted by a constructor. A lazy narrowing strategy rewrites expressions to head normal form,

The second idea is that we can combine all of the rules for each function into a definitional tree [9]. This is a pattern matching tree with some special properties. Definitional trees are relevant, because the current intermediate representation, FlatCurry, is a textual representation of a definitional tree.

As an example of a definitional tree, we use the append function from above.



While the needed narrowing strategy has been proven optimal in the number of rewrite steps, there is a lot more we can do to ensure that each rewrite step is computed efficiently. There has been some attempts to use other compilers to optimize Curry programs. Unfortunately, this has led to mixed results.

3 Need for an Optimizing Compiler

Before I describe how I plan to optimize Curry, I need to answer a fundamental question. Why bother? Is there a real need for optimizing Curry? There are currently three Curry compilers that have their own strategy for optimization. Pakcs compiles Curry to Prolog, and relies on the Prolog interpreter to optimize the logic programming code. The Kics2 compiler compiles Curry to Haskell, and relies on GHC to optimize the code. Finally, Sprite compiles Curry to LLVM, and uses LLVM to optimize the low level code. Each of these compilers has a significant disadvantage that could be solved by optimization.

Let's consider the Pakcs compiler. Pakcs will compile each Curry function into a set of Prolog clauses. This may seem like a problem, since Prolog doesn't have a concept of functions, but it turns out we can emulate a function very easily in Prolog. The trick is to add another parameter to the clause to represent the return value. I'll use the Curry function for adding all of the values in a list.

```
sum [] = 0
sum (x:xs) = x + sum xs
```

Let's translate this to Prolog. In Prolog, a clause doesn't return a value. It just finds values for variables such that the clause is true. So, our clause is a relation between a list and its sum.

```
sum(List, N)
```

we want to find a value for `N` such that this relation holds. We can do this with a recursive relation:

```
sum([],     0).
sum([X|XS], N) :- sum(XS,M), N is X+M.
```

This is nice and simple, unfortunately a Curry compiler can't use this translation because of lazy evaluation. Conside the `map` function:

```
map f [] = []
map f (x:xs) = f x : map f xs
```

Assuming we have an `apply` predicate that will apply a function to its argument, this would be translated as:

```
map(F, [],     []).
map(F, [X|XS], [FX|FXS]) :- apply(F,X,FX), map(F,XS,FXS).
```

Since we must apply `F` to `X` for this predicate, we will perform eager evaluation. All of the values of the list will be fully evaluated. To get around this problem, we must create a new type of node that is an unevaluated function application. `apply_F(F,X)`. This is similar to a thunk in Haskell. I'll use `apply_F` to represent an unevaluated application of a function. Now map will actually be translated to:

```
map(F, [],     []).
map(F, [X|XS], [apply_F(F,X) | apply_F(apply_F(map,F), XS)]).
```

Here we reconstructed the list, and created four new nodes, a new list, and three application nodes. This seems inefficient, but this process of rewiring graphs is very common in lazy languages. However, this has a negative effect on our `sum` example. This example must be translated to:

```
sum([],     0).
sum([X|XS], apply_F(apply_F(+_F, X), apply_F(sum_F, XS)).
```

We created three new nodes where the original translation didn't need any new nodes.

Creating new nodes is a problem, however this is not new. Haskell compilers have dealt with this issue for years, and they have several solutions. This leads to a new question. Could we compile Curry to Haskell and use a Haskell compiler to optimize the code? This is the idea behind the Kics2 compiler. This is a good idea in theory, but it has one major drawback. In order to compile Curry to Haskell we need to handle non-determinism in Haskell. This is not as straight forward as we might hope [**?**, 72]. A current approach is known as pull tabbing [11]. This strategy will be defined later, but it requires that every variable contains a unique identifier, and that we keep record of all of the non-deterministic choices that have been made.

The goal of the Kics2 compiler is that deterministic Curry functions are translated into simple Haskell functions that may contains a few extra parameters. Non-deterministic function may require more care. However, there are a few immediate consequences. If Kics2 cannot determine if a function is deterministic, then it must assume that it is not. Furthermore, GHC cannot inline any function that Kics2 compiled as non-deterministic. This is a problem because non-deterministic functions are significantly more expensive than deterministic functions.

This is unfortunate, but it is not entirely surprising. However, there are more severe consequences for compiling to Haskell. GHC will often fail to optimize deterministic functions. As an example consider the simple non-deterministic expression:

```
e = (1 + 2 + 3 + 4 + 5) `div` (1 ? 0)
```

It's clear that this expression only has one value. The expression `1 + 2 + 3 + 4 + 5 `div` 0` will fail. So, we should be able to optimize this to `15`. Unfortunately Kics2 is not able to optimize this, because GHC doesn't know what the `?` function is. Furthermore, it doesn't know how to optimize failure. This is expected, but we might expect GHC to be able to optimize the subexpression `1 + 2 + 3 + 4 + 5`. This is a simple case of constant folding.

In order to determine Kics2's ability to optimize Curry, it is necessary to look at the generated code. The Haskell code is not encouraging.

```
nd_C_test :: IDSupply -> Cover -> ConstStore -> Curry_Prelude.C_Int
nd_C_test s cd cs = let s0 = s
  in s0 `seq` Curry_Prelude.d_C_div (Curry_Prelude.d_OP_plus
  (Curry_Prelude.d_OP_plus (Curry_Prelude.d_OP_plus
  (Curry_Prelude.d_OP_plus (Curry_Prelude.C_Int 1#)
  (Curry_Prelude.C_Int 2#) cd cs) (Curry_Prelude.C_Int 3#) cd cs)
  (Curry_Prelude.C_Int 4#) cd cs) (Curry_Prelude.C_Int 5#) cd cs)
  (Curry_Prelude.nd_OP_qmark (Curry_Prelude.C_Int 0#)
  (Curry_Prelude.C_Int 1#) s0 cd cs) cd cs
```

This includes a lot of name mangling, which isn't surprising. However, what is surprising is the introduction of the `C_Int` constructor. We would hope that `d_OP_plus` and `C_Int` are wrappers for Haskell's `+` and `Int` respectively. Taking a look at the generated assembly code quickly confirms that this is not the case. In fact, GHC must construct a closure for each argument of each function, and it is not able to optimize anything in this expression.

At this point, it's reasonable to write this off as a bad example. Clearly no one would want to write this code. Even if they did, they could write it in a much more Haskell-friendly way.

```
fifteen = (1 + 2 + 3 + 4 + 5)
e = fifteen `div` (1 ? 0)
```

This is arguably better code than the first example, and GHC should be able to optimize `fifteen`. At this point it shouldn't be surprising that GHC fails the code generated by Kics2. The generated assembly code is actually very similar to the first example, except that `fifteen` is given its own closure. So, why can't GHC optimize either of these two examples? The problem is simple. `C_Int` isn't `Int`, and `d_OP_plus` isn't `+`. The GHC optimizer has no idea that these are constant arithmetic values, so it cannot assume that it can apply constant folding. We might hope that unboxing and inlining would be enough to get out

expression into a form that GHC can optimize, but the `Cover` and `ConstStore` arguments change the shape of the function so that GHC can't convert it to a form where constant folding would apply. Even though the expresion `1 + 2 + 3 + 4 + 5` is entirely deterministic, it might exist in a non-deterministic context. Therefore kicks2 needs to keep track of information for pull-tabbing.

There are similar problems with function inlining, and partial evaluation. However, I believe that this is the simplest example that demonstrates the problem. These compilation strategies will have a difficult time optimizing many common Curry programs. I believe that the best solution here is to build up the theory of optimization for functional logic programs. Much of this theory will be similar to optimizing imperative, functional, and logic programs, so it is worth taking a detour to explore the history of compiler optimizations.

4 Functional Compilers

There is a long history of functional programming with many different languages. However, my concern here is strictly the development of functional compiler technology, so I will instead focus on the compilers and abstract machines used to run functional languages.

The first real implementation of a functional language was LISP [61]. This was implemented with an interpreter. While it would late be compiled to more efficient versions [2, 41], LISP was not considered very practical.

Further implementations would compile a functional language to an abstract machine. This filled the role of an intermediate representation for functional languages. Landin's ISWIM language introduced an abstract machine known as the SECD machine [58]. This machine was an attempt to define the basic operations of a functional language. Further improvements were made by Felleisen and Friedman, who produced the CEK machine which is still used in the Racket programming language [34].

While strict functional languages were developing optimizing compilers, there was a call for lazy functional languages [52]. The problems raised by lazy evaluation lead to several new approaches to compilers and intermediate representations. One of these approaches was to compile to supercombinators, or functions that contain no free variables. The most influential version was the TIM (Three Instruction Machine) [33]. Although this was easier to implement, it is commonly believed to be inefficient compared to other machines.

The final approach to compiling lazy functional languages is to treat it as a problem in graph rewriting. While naive implementations of graph rewriting compilers often resemble interpreters, there are many optimizations that can be made to improve performance. The most prominent graph rewriting compilers is the graph reduction machine, or g-machine [57], which was used in the Miranda language. Two optimizations of the g-machine are removing the spines of functions, and removing tags identifying nodes in the graphs. The spine of a function is a sequences of function application nodes for functions with more than one argument. These are very common, and add several steps to evaluating

any function. Together the two optimizations resulted in the spineless tagless G-machine, which is used in the Glasgow Haskell Compiler [1, 65].

Along with abstract machines, there were several advancements for optimizing programming languages. Several optimizations, such as constant folding, could be translated immediately to functional languages. Several more optimizations came from early scheme compilers: Steel's Rabbit compiler, and Kranz' Orbit compiler [2, 41]. Steel showed how several common constructs in programming languages could be translated into lambda calculus [39, 43], which allowed for lambda calculus to function as an intermediate language that could be optimized. Later Steel showed how lambda calculus could be implemented efficiently [40, 42].

One of the optimizations in Rabbit and Orbit was the use of Continuation Passing Style as an intermediate representation. This allowed several dataflow optimizations to be performed more easily, and is similar to SSA for imperative languages [20]. Later, Flanagan and Felleisen, among others, refined CPS by removing "administrative redexes" to Administrative Normal Form [35]. This representation is commonly used in modern functional compilers, and is the basis for the STG-machine [66].

5 Logic Compilers

Logic programming has largely been more concerned with issues of completeness and soundness rather than efficiency. While there are optimizing compilers for logic languages, most of the research is concerned with semantics and features.

While there were logic systems such as QA3 and Planner [56], the first real logic programming language was Prolog [26]. Prolog, and logic languages in general, are distinct from other programming languages because they handle control flow themselves. There is debate about whether this is the correct way to program, or if the user should specify the control [51, 56], but this is not relevant for this thesis.

While optimizing compilers for Prolog existed [23], most modern implementations of Prolog use Warren's Abstract Machine (WAM) [74] to define semantics, and optimizations for Prolog [27,49]. It is notable that the WAM consists of instructions to manipulate graphs in a heap, procedure calls, and backtracking.

The Mercury language added types, and the ability to control non-determinism and free variable instantiation [70]. This control of free variables, which they call modes, allows for many optimizations that aren't possible in Prolog [70]. The Oz language focuses on Data Parallelism and constraint programming. While Oz is a logic programming language, it is meant to be a multi-paradigm language. As such, it is not surprising that its implementations tend to be slower than Mercury [69, 70]

6 Functional Logic Compilers

While there have been many attempts to add functions to logic programming languages, the idea of combining the two paradigms with a single formalism is fairly recent. Most previous attempts were focusing on either adding functions to logic programming languages, or adding non-determinism and logic variables to functional programming languages. This often involved transforming one set of features into another. While this can work, the theory is messy.

Modern functional logic languages attempt to unify the two concepts from the ground up. Functional logic programs are thought of as term, and graph, rewrite system. Computation in functional logic programs is done through narrowing. There has been a lot of research into devising an optimal narrowing strategy; however, there has been little work done on optimizing programs themselves.

One of the first functional logic languages is ALF [46]. ALF integrates functional and logic programming using a graph rewrite system. Computations in ALF use innermost narrowing, which is similar to eager evaluation. ALF also had several restrictions. For example, functions had to be confluent and terminating. ALF compiled to a variant of the WAM, although there was no attempt at making ALF-specific optimizations.

KLEAF took a different approach, and compiled to a flat intermediate language, and then used SLD-resolution to compute answers [37]. The advantage is that KLEAF was able to compute more programs than ALF could. In addition to the programs of ALF, KLEAF could also compute non-terminating, locally confluent systems. It also had very limited support for lazy evaluation.

Babel introduced lazy evaluation to functional logic programming [44]. Similar to ALF, babel programs were computed with graph rewriting. However, babel compiled to an abstract graph rewriting machine.

The LIFE programming language combined logic programming, functional programming, and object oriented programming. [4] While the object oriented programming isn't relevant, life also introduced a method for suspending evaluation of equations until enough information had been determined to resolve the equation. This method, known as residuation, is one of the features of Curry.

The Toy language was an attempt to construct a lazy functional logic language [25]. Syntactically Toy is similar to Prolog, and it uses a lazy narrowing strategy. Toy is similar to Curry in many respects, but it is no longer under active development.

7 Compiling Curry

Pakcs, Kics2, and Sprite all have their own strategy for compiling Curry. While these compilers all compile to different languages, the more important difference is how they handle non-determinism. In this section I'll take an in-depth look at these compilers, and use these ideas to propose an adequate compilation strategy to demonstrate optimizations.

7.1 Pakcs

The Pakcs compiler is built on the foundations of Prolog. Not only does Pakcs compile Curry to Prolog, but it does so in such a way that all of the non-standard features of Curry are handled by Prolog. For example, free variables in Curry are translated to free variables in Prolog, and the choice operator in Curry is translated into parallel clauses in Prolog. The immediate consequence of this is that all non-determinism is done through backtracking. While this strategy is efficient, it is not complete. There are Curry programs that could produce an answer, but will fail to do so in Pakcs.

There are several advantages to translating Curry into Prolog. One advantage is that the translation is very simple. A compiler from FlatCurry to Prolog can be written over the course of a week. Another advantage is the entirely reasonable assumption that the Prolog compiler will do a better job of implementing non-determinism and free variables than a Curry compiler would. The basis for this assumption is that more research and work has gone into the Prolog compiler. Furthermore, any improvements of the efficiency of Prolog directly correlate to improvements in the efficiency of Curry. While these benefits are nice, I have already discussed the drawbacks of compiling to Prolog.

I will demonstrate the translation from Curry to Prolog using the `sum` example from before.

```
sum [] = 0
sum (x:xs) = x + xs
```

Below is a simplified version of the code generated by Pakcs. For simplicity I am removing code related to residuation [48].

```
sum(Arg,Result) :- hnf(Arg, HArg), sum_1(HArg, Result).

sum_1([],      0).
sum_1([X|XS],  Result)  :- !, hnf(+_F(X, sum_F(XS)), Result).
sum_1(fail(F), fail(F)) :- nonvar(F).
```

Those familiar with Prolog may find this a little odd. Why are we calling a mysterious `hnf` predicate to compute the value of `X + sum(XS)`? In fact, why do we have two separate `sum` predicates at all? Those familiar with functional interpreters may have already guessed where this is headed. Pakcs doesn't compile Curry programs, it generates rules that interpret Curry expressions. There are two rules generated, `nf` and `hnf`, which are responsible for reducing expressions to normal form and head normal form respectively.

The `nf` rule is actually static across all programs. It computes the head normal form of the argument, and computes the normal form of all of the argument's children. The `hnf` rule must be generated for every new program. It is essentially a giant case discriminator for every possible function symbol. As an example:

```
hnf(sum_F(XS), Result)        :- !, sum(XS,Result).
hnf(foldl_F(OP,Z,XS), Result) :- !, foldl(OP,Z,XS, Result).
hnf(++_F(XS,YS), Result)      :- !, ++(XS, YS, Result).
hnf(+_F(X,Y), Result)         :- !, +(X, Y, Result).
```

Here each function is compiled in a manner similar to `sum`. There are a few inherent inefficiencies in this style of compilation. First, we construct several nodes that we don't need. For example, if we evaluate the expression `sum [2]` we will produce the intermediate result. `+_F(2, sum_F([]))`. There is no reason to construct the `+` node. Second, every expression that does not directly produce a value makes a call to the `hnf` predicate. This call to `hnf` is inherently slow, due to the fact that it contains a giant case discriminator. In fact, it functions similarly to the giant switch statement in byte-code interpreters [32].

7.2 Kics2

The Kics2 compiler takes a different approach, and compiles Curry to Haskell. The hope is that deterministic functions can be compiled to Haskell code, and therefore take advantage of GHC's many optimizations. Unfortunately, as we have already seen, this doesn't quite work out.

Again, I'll demonstrate Kics2's compilation strategy by example. Consider the sum function again. I will simplify the generated code considerably for the sake of readability. Sum will generate the following Haskell code:

```
sum :: OP_List C_Int -> Cover -> ConstStore -> C_Int
sum x1 cd cs = case x1 of
  List -> C_Int 0#
  Cons x2 x3 -> c_plus x2 (sum x3 cd cs) cd cs
  Choice d i l r -> narrow d i (sum l cd cs) (sum r cd cs)
  Fail_List d info -> failCons d (traceFail "sum" [show x1] info)
```

Here we implement `sum` as a case statement. Every algebraic type in Curry is augmented with two possibilities, `Choice` and `Fail`, so every function must check those two cases. If the constructor is the empty list, then we return 0 as normal. If the constructor is not empty, then we call the `c_Plus` function defined in Kics2. However, if we encounter a choice, we cannot proceed. In this case we move the choice up one level in our expression, and compute each choice independently. This is called pull-tabbing [8, 11]. As an example the expression `sum ([1] ? [2])` will match the `Choice` case, and return `Choice cd i (sum l cd cs) (sum r cd cs)`. Note that `(sum l cd cs)` and `(sum r cd cs)` will not be evaluated due to Haskell's laziness.

There is a non-obvious consequence of this compilation strategy. Every function that pattern matches in Kics2 contains a case for `Choice`. Since `Choice` is a constructor, it is a head normal form in Haskell. Therefore, if a choice is ever generated, it will immediately percolate up to the root of the expression. This will copy every node above the node in the expression.

7.3 Sprite

The final Curry compiler is Sprite [18]. Sprite compiles Curry to LLVM assembly code, and uses a pull tabbing scheme. The difference between Sprite and Kics2 is that Sprite uses the Fair Scheme [17] for executing choices. The preliminary results for Sprite look promising. Unfortunately, since Sprite isn't complete, I can't make any meaningful comparisons to other compilers.

7.4 Graph Rewriting vs Graph Reduction

These compilers show the two different ideas of compiling lazy functional programs. Pakcs compiles all of the Curry functions into rewrite rules. Expressions are rewritten until they reach a normal form. The alternative to graph rewriting is graph reduction. The idea is that the graph knows how to reduce itself to a normal form. This is the approach commonly taken by lazy functional language compilers. Each node in the graph contains a pointer to code that will compute a value in head normal form for that node. These nodes with code pointers are commonly known as closures.

While there is nothing showing that graph rewriting is inherently slower than graph reduction, the common belief among the functional community is that rewriting is slower. There is a possibility that, with aggressive optimizations, graph rewriting may be competitive with graph reduction, but that is not the focus of this research.

7.5 Compilation

In order to optimize Curry, I need to have a well founded compilation model. This model has two main goals: simplicity, and efficiency. I do not claim that this is the best model for Curry, but it is easy to understand, and reasonably efficient.

First I need to answer some fundamental questions about this model. The first question to ask is, "How should I represent the abstractions in Curry?" There are two abstractions in Curry that come from logic programming: non-determinism and free variables. While I could handle these independently, it is well known that free variables can be converted into non-determinism and vice versa. For example, in a language with only free variables I can construct the choice operator as follows:

```
(?) :: a -> a -> a
x ? y = if b then x else y
 where b free
```

Furthermore, if I only have non-determinism, then I can construct values using a generator for each type:

```
freeBool :: Bool
freeBool = True ? False
```

The expression `b where b free` could be translated to `freeBool` by the compiler. I have no strong reason to prefer one of these representations over the other. In fact, both of these representations have been proposed [15, 24]. I plan to translate free variables to non-determinism for two reasons: this is currently the standard representation of Curry programs, and representing free variables explicitly will likely require some clever trick such as variant heaps [15]. This would complicate the compiler.

7.6 Non-determinism

The second task is to determine how to represent non-determinism This has recently been the subject of a lot of research. Currently there are four options for representing non-determinism. Backtracking, Copying, Pull-tabbing, and Bubbling. All of these options are incomplete in their naive implementations. However, all of them can be made complete. [17]

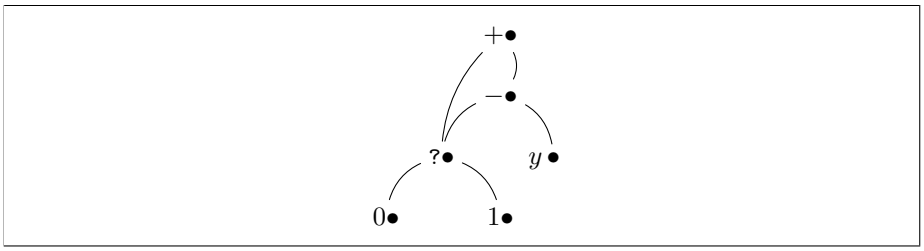
Backtracking is conceptually the simplest mechanism for non-determinism. We evaluate an expression normally, and every time we hit a choice operator, we pick one option. If we finish the computation, either by producing an answer or failing, then we undo each of the computations until the last choice expression. We continue until we've tried every possible choice.

There are a few issues with backtracking. Aside from being incomplete, a naive backtracking implementation relies on copying each node as we evaluate it, so we can undo the computation. Solving incompleteness is a simple matter of using iterative deepening instead of backtracking. This poses its own set of issues, such as how do we avoid producing the same answer multiple times, however these are not difficult problems to solve. The issue of copying every node we evaluate is a bigger issue, as it directly competes with any attempt to build an optimizing compiler. However, I believe that we can avoid creating many of these backtracking nodes.

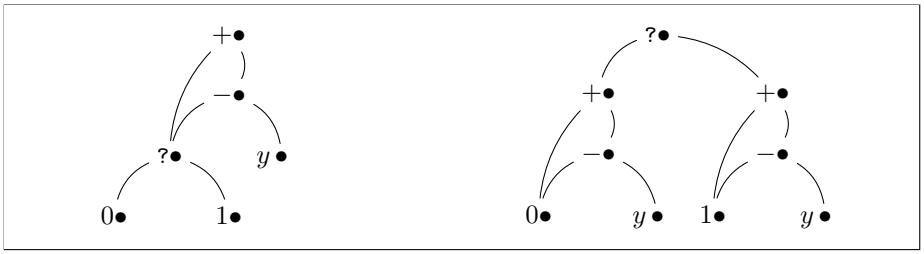
The following three mechanisms are all based on the idea of copying part of the expression graph. All of them are incomplete with a naive implementation, however they can all be made complete using the fair scheme [**?**]. I'll demonstrate each of these mechanisms with the following expression.

```
x + y
 where x
```

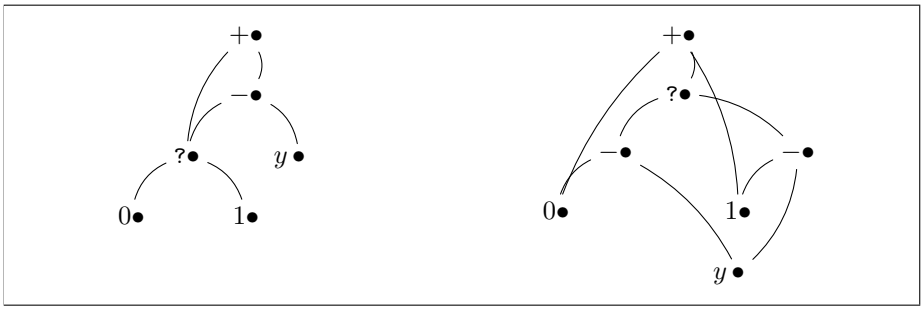
This expression has the following graph:



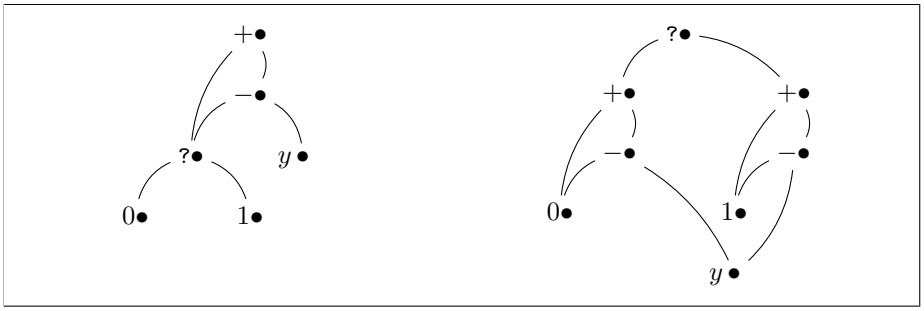
Copying is a different take on non-determinism. The idea is straightforward. Any time we come to a choice node we make a copy of the graph for each choice. Then we going the copies with a new choice node. Below I give an example of what the graph looks like before and after copying for the expression:



Pull-tabbing is the other extreme for moving non-determinism. Instead of moving the choice node to the root of the graph, we move the choice node up one level. [8] A naive implementation of pull-tabbing isn't even valid, so identifier must be included for each variable to represent which branch it is on. There is a significant cost to keeping track of these identifiers.



Bubbling is a more sophisticated approach to moving non-determinism. Instead of moving the choice node to the root, we move it to it's dominator. [12] Bubbling is always valid, and we aren't copying the entire graph. Unfortunately computing dominators at runtime is expensive. There are strategies of keeping track of the current dominator, [13] but as of this time, there are no known bubbling implementations.



There is one major issue that I've swept under the rug. So far every Curry implementation use graph rewriting as opposed to graph reduction. Kics2 comes

the closest, be all non-deterministic expressions are copied all the way to the root. Graph reduction and non-determinism are both well understood individually, however there is an issue when trying to combine the two. As an example, we'll look at graph reduction with backtracking.

In a graph rewriting system backtracking is straightforward. If node l rewrites to node r, then we push (l, r) onto a global backtracking stack. When it comes time to backtrack we simply pop each (l, r) pair from the stack, and replace the node r with l until we pop a node where l was a choice. At this point the graph is in the same state it was before the choice was taken.

While this works great, the whole point of using graph reduction over graph rewriting is to avoid constructing intermediate nodes when evaluating an expression to head normal form. If a node isn't constructed, then we can't backtrack to it. One solution to this problem is to identify a set of nodes that must be copied for backtracking. It is likely impossible to implement non-determinism without creating and copying some nodes, but this compiler should aim to minimize that copying.

7.7 Compiler Pipeline

Since I am demonstrating the value of optimizations, I can reuse existing pieces of Curry compilers. Currently, every Curry compiler produces an intermediate representation known as FlatCurry. A FlatCurry program represents a Curry program, after the transformation to definitional trees has been made.

FlatCurry is similar to the Core language in GHC. There are new cases for free variables and non-deterministic expressions. A partial specification of FlatCurry is given below.

$$
\begin{array}{lcl}
Prog & = & Prog\ name\ imports\ Type^*\ Function^*\ Op^* \\
\ldots & & \\
Function & = & Func\ name\ arity\ visibility\ Type\ Rule \\
Rule & = & Rule\ var^*\ Expr \\
Expr & = & Var\ var \\
 & | & Lit\ literal \\
 & | & Comb\ CombType\ name\ Expr^* \\
 & | & Let\ (var, Expr)^*\ Expr \\
 & | & Free\ var^*\ Expr \\
 & | & Or\ Expr\ Expr \\
 & | & Case\ CaseType\ Expr\ (Pattern^*, Expr)^* \\
CombType & = & FunctionAp\ |ConstructorAp \\
 & | & PartFuctionAp\ |PartConstructorAp \\
Pattern & = & Pattern\ name\ var^* \\
 & | & LPattern\ literal
\end{array}
$$

While FlatCurry is an intermediate representation, it is not well suited to some optimizations. An important part of this dissertation will involve developing an IR that is suitable for optimizing Curry programs. An appropriate

IR should represent the basic operations to evaluate a program. In imperative languages, IRs are similar to assembly code. In functional languages, IRs are variants of lambda calculus, where the fundamental operation is function application. In logic languages, the WAM uses replacement and choices as its fundamental operations. While I might want to use a combination of lambda calculus and the WAM, this does not seem like the right abstraction. Furthermore, the WAM is designed for a backtracking system, so this would not be a general solution.

Instead, I propose an IR based on graph rewriting. This is not a new idea [21,38,65]. In fact, this was one of the major innovations behind the G-machine. While the STG-machine is an IR for a graph reduction machine, it is still framed in terms of lambda calculus. One possibility is that the IR is a variation on ICurry, an imperative language for representing curry programs. [18]. ICurry is fundamentally a language for rewriting graphs, but it has been developed specifically with Curry programs in mind. Currently the IR is not able to incorporate all of the optimizations. However, I believe that this IR is a good starting place for optimizing Curry.

8 Optimizations

Now that I have a general compilation strategy, I need to apply optimizations. As previously mentioned, there are many optimizations to choose from.

The theory of compiler optimizations is nearly half a century old. Most commonly applied optimizations today originated back in 1969 [5]. In 1972 Allen et al. cataloged several optimizations, and showed how to use data flow analysis for optimization [6]. At this point all optimizations were local to a given block of code. In 1973 Kindall showed how to do global program optimization using dataflow analysis [54]. Alpern et al. developed the SSA intermediate representation where variables are never reassigned [7]. Wegman et al. showed several optimizations that took advantage of SSA [75]. Stolts et al. used induction analysis to further optimize program [71]

8.1 Problems With Traditional Optimizations

While these optimizations are all useful, they may not be applicable to Curry. As an example, one very prevalent optimization from functional programming is function inlining. This is where we take function call and, after appropriately renaming variables, replace the call with the function body. Inlining reduces the number of function calls we need to make, and gives us more code to make other optimizations. This seems like a good candidate. Unfortunately, this doesn't work well with non-determinism. Suppose we have the code:

```
xor True True   = True
xor True False  = False
xor False True  = False
xor False False = True
```

```
xorSelf x = xor x x

xorSelf (True ? False)
```

It should be the case that `xorSelf` will always return `False`, but if we inline the line `xorSelf (True ? False)` we will be left with `xor (True ? False) (True ? False)`. This expression can now evaluate to `True`. This violates the call-time choice semantics of Curry [59]. Therefore, I cannot use simple textual replacement to implement inlining or β-reduction as in Haskell or ML. This is another point in favor of using a graph-based IR.

We run into further problems with traditional optimizations. The biggest problem is that they may not be effective. Constant folding is a useful optimization for programs with a large amount of arithmetic, but it may not be as effective in Curry. I should be careful to avoid implementing optimizations that will not be effective.

To this end I am interested in optimizations that are going to be effective for Curry. One major problem with Curry performance is memory usage. As we have seen already, many Curry problems require much more space than they should. This, in turn, prompts Curry programs to spend a lot of their running time creating and destroying small pieces of memory. One way to improve Curry performance is to reduce the number of nodes that are created and destroyed. To that end, I've picked three optimizations to implement: unboxing, deforestation, and shortcutting.

8.2 Unboxing

Boxing refers to abstracting primitive data types in order to make a uniform representation of data. This is a very common process in programming languages. It makes development of the compiler easier, and it reduces hard-to-find errors. This is more important in lazy languages where a variable or parameter may be represented by unevaluated code instead of a concrete value. However, boxing has a major performance penalty. All arithmetic with boxed values must first dereference the values. In order to alleviate this performance penalty we would like to unbox values. Unboxing is the process of replacing boxed literals with literal values. Launchbury et al. showed that unboxing can be implemented in a lazy functional language [53]. Hall et al. showed that unboxing can be implemented using partial evaluation [45]. It remains to be shown that unboxing can be implemented with non-deterministic values. This seems reasonable, since non-deterministic values can behave like unevaluated expressions.

8.3 Deforestation

Deforestation is an optimization technique that originated with functional programming [73]. The idea is that there are many functional programs that produce intermediate data structures, and immediately consume them. Deforestation will rewrite these functions so that they do not create these data structures.

Wadler showed the initial deforestation transformation [73]. While this transformation removed all intermediate structures, it was only applicable to a limited set of programs. Launchbury et al. showed how this transformation could simplified and expanded to a wider class of programs [36]. Unfortunately this transformation isn't guaranteed to remove every intermediate structure. Currently, the standard deforestation technique is known as stream fusion. This has been shown to be more effective than shortcut deforestation, but is requires re-writing much of the standard library [28]. While the implementation of deforestation should be straightforward, it remains to be shown the deforestation is still valid for non-deterministic expressions.

To see how deforestation works, consider the code for `all`:

```
all p xs = and (map p xs)
```

While this code is correct, it's not optimal. If we start with the list `[x0, x1, ... xn]`, then `map p` will produce the intermediate list `[p x0, p x1, ... p xn]`. Then finally we fold all of those values with `and`.

We could instead write the code:

```
all p xs = h xs
 where h [] = True
       h (y:ys) = p y && h ys
```

This is less obviously correct, but it is more efficient. We won't create the intermediate list. We would like the compiler to automatically transform the first version into the second version. We can accomplish this by recognizing that `and` and `map` can be implemented using `foldr` and `build` respectively.

```
and = foldr (&&) True
map f xs = build (\c n -> foldr (\a b -> c (f a) b) n xs)
```

Then we can apply the rule `foldr f z (build g) = g f z`. This will produce the code

```
all p xs = foldr (&&) True (build (\c u -> foldr (\a b -> c (p a) b) n xs))
         = (\c n -> foldr (\a b -> c (p a) b) n xs) (&&) True
         = foldr (\a b -> (p a) && b) True xs
```

Substituting the definition for `foldr` we see that `all` reduces to:

```
all p xs = h xs
 where h [] = True
       h (y:ys) = (p a) && h ys
```

This is the code we wanted originally.

8.4 Shortcutting

Shortcutting is a relatively recent idea for optimizing functional logic programs [16]. It is similar to the idea of deforestation, in that we are avoiding the construction of intermediate data structures. However, the goal of shortcutting is to avoid the construction of intermediate function nodes. Shortcutting has been shown to be effective in theory, but has yet to be implemented in a full compiler.

The idea is straightforward, however is it dependent on the compilation strategy for Curry. When a function in Curry is compiled, code is generated to reduce that function symbol to head normal form. Usually this code is uniform, and we represent it with the function **H**.
As an example:
$\mathbf{H}(length([])) = 0$
$\mathbf{H}(length(x : xs)) = \mathbf{H}(+(1, length(xs)))$
$\mathbf{H}(length(xs)) = \mathbf{H}(length(\mathbf{H}(xs)))$

$\mathbf{H}([] +\!\!+ []) = []$
$\mathbf{H}([] +\!\!+ (y : ys)) = y : ys$
$\mathbf{H}([] +\!\!+ ys) = \mathbf{H}(ys)$
$\mathbf{H}((x : xs) +\!\!+ ys) = x : (xs +\!\!+ ys)$
$\mathbf{H}(xs +\!\!+ ys) = \mathbf{H}(\mathbf{H}(xs) +\!\!+ ys)$

It's clear that we're creating more nodes then we need to. For example, in the second rule for $\mathbf{H}(length)$ we create the node + only to immediately evaluate it. It would be beneficial if we could just execute the code for + directly. We accomplish this in two steps. First we generate a separate reduction function for each type of node, and second we allow a node to call other reduction functions. The effect of this is that we don't need to construct nodes that we will immediately reduce. Instead, we just call their reduction function.

For example, the compiled code for `length` would become:
$\mathbf{H}_{length}([]) = 0$
$\mathbf{H}_{length}(x : xs) = \mathbf{H}_{+}(1, xs)$
$\mathbf{H}_{length}(xs) = \mathbf{H}_{length}(\mathbf{H}(xs))$

The justification for this transformation is given in [16]. This transformation has been shown to speed up programs significantly over the naive version. I also expect this optimization to work well with other optimizations.

It should be mentioned that other lazy functional languages can get an effect similar to this, but it requires either inlining the function call, or separate rules for compiling built in functions [65]. The shortcutting approach is conceptually simpler, and more general. However, it comes at the cost of the size of the generated code.

All three of these optimizations are focused on saving memory at runtime. While there are many other possible optimizations, I believe that these will be effective in a real Curry compiler.

9 Deliverables

In order to demonstrate that I have completed the requirements for this presentation, I will deliver three products . First, I will deliver an optimizing Curry compiler that implements the optimizations described in this paper. While the compiler itself will be a physical product, I also intend to provide a logical framework in which it is easy to write and test new optimizations. Second, I will prove the correctness of each optimization. Finally, I will demonstrate the efficiency of the compiler by comparing it to both an unoptimized compiler and current Curry compilers.

While I believe the first two parts are self-explanatory, the final part will require more care. Determining the efficiency of an optimization can be a tricky task, as Mytkowicz et al. showed [62]. There is a possibility that we could use the STABILIZER platform [29] to alleviate these issues Aside from the issues that Mytkowicz et al. raise, I also need to demonstrate that the optimizations are effective.

While I could just compare the performance to standard Curry compilers, I also plan to test each optimization in combination. This will tell me which optimizations are effective, and which are redundant. As an example, it may be that deforestation and unboxing both optimize the same cases, so, while it looks like they both provide a significant improvement, they may not perform as well in combination.

9.1 completed work

Writing an optimizing compiler for any language is a daunting task. In order to complete this project in a reasonable timeframe I need a plan. Fortunately I've already completed a substantial amount of work.

So far my most substantial contribution has been identifying areas in the Pakcs and Kics2 compiler that are inefficient. This is important for identifying what optimizations will be most beneficial. I have also developed a graph reduction strategy for Curry programs, and a transformation from FlatCurry to C code. This strategy is entirely correct for deterministic programs, and I believe it can be extended to non-deterministic programs, although this remains to be demonstrated.

While these steps are important, they are all entirely illusory. There is no code for them, they are all just an idea of how to compile Curry programs. In a more substantial sense, I have developed a rewriting engine for FlatCurry programs. This engine is simple to use. An optimization can be represented as a function from `Expr -> Expr`. The engine is also powerful enough to represent complicated optimizations such as inlining. For an example, A constant folding pass can be written with the code:

```
constFold e | pattern e "==" [lit a, lit b] = (boolc (a == b))
 where a, b free
constFold e | pattern e "+" [ilit a, ilit b] = (intc (a + b))
```

```
 where a, b free
...
```

9.2 Timeframe

There is still a substantial amount of work left to be done. I have broken down the remaining work into eight separate tasks with an expected time to completion.

- build a naive compiler: 3 months
- implement shortcutting: 2 months
- prove correctness of shortcutting with non-determinism: 1 months
- implement unboxing: 2 months
- prove correctness of unboxing: 1 month
- implement deforestation: 2 months
- prove correctness of deforestation: 2 months
- write dissertation: 9 months

I have tried to be overly pessimistic with this timeframe. This will be important if one section takes longer than I'm expecting.

10 Conclusion

Optimizations have been shown to be very effective at improving performance of compiled code. While it is a common belief that modern compilers can pass off optimization to a language with an optimizing compiler, I don't believe that is correct. I have shown that it is not enough to simply translate Curry to a language with an optimizing compiler. Instead, Curry needs optimizations specifically designed for it. In my dissertation I will be implementing three of these optimizations; shortcutting, backtracking, and unboxing. I will also be showing the correctness of each of these optimizations.

References

[1] The Glasgow Haskell Compiler. https://www.haskell.org/ghc/.

[2] Norman Adams, David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, and James Philbin. Orbit: An optimizing compiler for scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 219–233, New York, NY, USA, 1986. ACM.

[3] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design (Addison-Wesley Series in Computer Science and Information Processing)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1977.

[4] Hassan Aït-Kaci. An introduction to life-programming with logic, inheritance, functions, and equations. In *Proceedings of the 1993 International Symposium on Logic Programming*, ILPS '93, pages 52–68, Cambridge, MA, USA, 1993. MIT Press.

[5] F.E. Allen. Program optimization. *Annual Review in Automatic Programming*, 5:239–307, 1969.

[6] F.E. Allen and J. Cocke. *A catalogue of optimizing transformations*. Prentice-Hall, 1972.

[7] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 1–11, New York, NY, USA, 1988. ACM.

[8] A. Alqaddoumi, S. Antoy, S. Fischer, and F. Reck. The pull-tab transformation. In *Third International Workshop on Graph Computation Models*, page N/A, Enschede, The Netherlands, October 2010.

[9] S. Antoy. Definitional trees. In H. Kirchner and G. Levi, editors, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 143–157, Volterra, Italy, September 1992. Springer LNCS 632.

[10] S. Antoy. Optimal non-deterministic functional logic computations. In *Proceedings of the Sixth International Conference on Algebraic and Logic Programming (ALP'97)*, pages 16–30, Southampton, UK, September 1997. Springer LNCS 1298. Extended version at http://cs.pdx.edu/ antoy/homepage/publications/alp97/full.pdf.

[11] S. Antoy. On the correctness of pull-tabbing. *TPLP*, 11(4-5):713–730, 2011.

[12] S. Antoy, D. Brown, and S.-H. Chiang. On the correctness of bubbling. In F. Pfenning, editor, *17th International Conference on Rewriting Techniques and Applications*, pages 35–49, Seattle, WA, August 2006. Springer LNCS 4098.

[13] S. Antoy and S. Libby. Making bubbling practical. In *Proc. of the 27th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2018)*. Springer, 2011.

[14] Sergio Antoy, Rachid Echahed, and Michael Hanus. A needed narrowing strategy. *J. ACM*, 47(4):776–822, July 2000.

[15] Sergio Antoy, Michael Hanus, Jimeng Liu, and Andrew Tolmach. A virtual machine for functional logic computations. In Clemens Grelck, Frank Huch, Greg J. Michaelson, and Phil Trinder, editors, *Implementation and Application of Functional Languages*, pages 108–125, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[16] Sergio Antoy, Jacob Johannsen, and Steven Libby. Needed computations shortcutting needed steps. In *Electronic Proceedings in Theoretical Computer Science, EPTCS*, volume 183, pages 18–32. Open Publishing Association, 2015.

[17] Sergio Antoy and Andy Jost. Compiling a functional logic language: The fair scheme. In Gopal Gupta and Ricardo Peña, editors, *Logic-Based Program Synthesis and Transformation*, pages 202–219, Cham, 2014. Springer International Publishing.

[18] Sergio Antoy and Andy Jost. A new functional-logic compiler for curry: Sprite. *CoRR*, abs/1608.04016, 2016.

[19] Andrew W. Appel. *Modern Compiler Implementation in ML: Basic Techniques*. Cambridge University Press, New York, NY, USA, 1997.

[20] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, NY, USA, 2007.

[21] H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeijer, and M. R. Sleep. Towards an intermediate language based on graph rewriting. In J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, editors, *PARLE Parallel Architectures and Languages Europe*, pages 159–175, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[22] M. Bezem, J. W. Klop, and R. de Vrijer (eds.). *Term Rewriting Systems*. Cambridge University Press, 2003.

[23] D.L. Bowen, L. Byrd, F.C.N. Pereira, L.M. Pereira, and David Warren. Decsystem-10 prolog user's manual. Technical report, Department of Artificial Intelligence University of EdinburghSRI International, Nov 1982.

[24] B. Brassel. *Implementing Functional Logic Programs by Translation into Purely Functional Programs*. PhD thesis, Christian-Albrechts-Universität zu Kiel, 2011.

[25] I. Castieiras, J. Correas, S. Estvez-Martn, and F. Senz-Prez. Toy: A cflp language and system. *The Association for Logic Programming*, 2012.

[26] Alain Colmerauer and Philippe Roussel. The birth of prolog. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 37–52, New York, NY, USA, 1993. ACM.

[27] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The yap prolog system. *Theory Pract. Log. Program.*, 12(1-2):5–34, January 2012.

[28] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion. from lists to streams to nothing at all. In *ICFP07*, 2007.

[29] Charlie Curtsinger and Emery D. Berger. Stabilizer: Statistically sound performance evaluation. *SIGARCH Comput. Archit. News*, 41(1):219–228, mar 2013.

[30] Bart Demoen, Maria García de la Banda, Warwick Harvey, Kim Marriott, and Peter Stuckey. An overview of hal. In Joxan Jaffar, editor, *Principles and Practice of Constraint Programming – CP'99*, pages 174–188, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[31] R. Echahed and J. C. Janodet. On constructor-based graph rewriting systems. Technical Report 985-I, IMAG, 1997. Available at ftp://ftp.imag.fr/pub/labo-LEIBNIZ/OLD-archives/PMP/c-graph-rewriting.ps.gz.

[32] M. Anton Ertl and David Gregg. The behavior of efficient virtual machine interpreters on modern architectures. In Rizos Sakellariou, John Gurd, Len Freeman, and John Keane, editors, *Euro-Par 2001 Parallel Processing*, pages 403–413, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[33] Jon Fairbairn and Stuart Wray. Tim: A simple, lazy abstract machine to execute supercombinators. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, pages 34–45, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.

[34] Matthias Felleisen and Daniel P. Friedman. Control operators, the SECD-machine, and the lambda-calculus. pages 193–219, June 1986.

[35] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, 1993.

[36] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '93, pages 223–232, New York, NY, USA, 1993. ACM.

[37] Elio Giovannetti, Giorgio Levi, Corrado Moiso, and Catuscia Palamidess. Kernel-leaf: A logic plus functional language. *Journal of Computer and System Sciences*, 42(2):139 – 185, 1991.

[38] J. R. W. Glauert, J. R. Kennaway, and M. R. Sleep. Dactl: An experimental graph rewriting language. In Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Graph Grammars and Their Application to*

Computer Science, pages 378–395, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[39] Jr. Guy Lewis Steele. Lambda: The ultimate declarative. November 1976.

[40] Jr. Guy Lewis Steele. Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or lambda, the ultimate goto". *ACM Conference Proceedings. 1977*, 1977.

[41] Jr Guy Lewis Steele. *RABBIT: A Compiler for SCHEME*. PhD thesis, May 1978.

[42] Jr. Guy Lewis Steele. Compiler optimization based on viewing lambda as rename + goto. 1980.

[43] Jr. Guy Lewis Steele and Gerald Jay Sussman. Lambda: The ultimate imperative. March 1976.

[44] JJ Moreno-Navarro H Kuchen, R Loogen. The functional logic language babel and its implementation on a graph machine. In *New Generation Computing*, page 391427. Springer LNCS 4079, 1996.

[45] Cordelia Hall, Simon L. Peyton Jones, and Patrick M. Sansom. Unboxing using specialisation. In Kevin Hammond, David N. Turner, and Patrick M. Sansom, editors, *Functional Programming, Glasgow 1994*, pages 96–110, London, 1995. Springer London.

[46] M. Hanus and A. Schwab. Alf users manual. 1995.

[47] M. Hanus (ed.). Curry: An integrated functional logic language (vers. 0.9.0), 2016. Available at http://www.curry-language.org.

[48] M. Hanus (ed.). PAKCS 1.14.3: The Portland Aachen Kiel Curry System, March 04, 2017. Available at http://www.informatik.uni-kiel.de/ pakcs.

[49] Ralph Clarke Haygood. Native code compilation in sicstus prolog. In *Proceedings of the Eleventh International Conference on Logic Programming*, pages 190–204, Cambridge, MA, USA, 1994. MIT Press.

[50] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López, José F. Morales, and German Puebla. *An Overview of the Ciao Multiparadigm Language and Program Development Environment and Its Design Philosophy*, pages 209–237. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[51] Carl Hewitt. Planner: A language for proving theorems in robots. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, pages 295–301, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[52] Bloomington. Computer Science Department Indiana University, D.P. Friedman, and D.S. Wise. *CONS SHOULD NOT EVALUATE ITS ARGUMENTS.* Indiana University. Computer Science Department, 1975.

[53] SL Peyton Jones, J Launchbury, and Simon Peyton Jones. Unboxed values as first class citizens. In *ACM Conference on Functional Programming and Computer Architecture (FPCA'91)*, volume 523, pages 636–666. Springer, January 1991.

[54] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[55] J. W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 1–112. Oxford University Press, 1992.

[56] Robert Kowalski and Keith L. Clark. Logic programming. In *Encyclopedia of Computer Science*, pages 1017–1031. John Wiley and Sons Ltd., Chichester, UK.

[57] Augustsson L. *Compiling Lazy Functional Languages.* PhD thesis, 1978.

[58] Peter Landin. The mechanical evaluation of expressions. *The Computing Journal*, pages 308–320, 1964.

[59] F. J. López-Fraguas, E. Martin-Martin, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and narrowing for constructor systems with call-time choice semantics. *TPLP*, 14(2):165–213, 2014.

[60] Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, January 1969.

[61] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[62] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing wrong data without doing anything obviously wrong! *SIGPLAN Not.*, 44(3):265–276, March 2009.

[63] Bjrn Peemller. *Normalization and Partial Evaluation of Functional Logic Programs.* PhD thesis, 2016.

[64] Simon Peyton Jones and Simon Marlow. Secrets of the glasgow haskell compiler inliner. *J. Funct. Program.*, 12(5):393–434, July 2002.

[65] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[66] Simon L. Peyton Jones and Jon Salkild. The spineless tagless g-machine. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 184–201. ACM, 1989.

[67] D. Plump. Term graph rewriting. In H.-J. Kreowski H. Ehrig, G. Engels and G. Rozenberg, editors, *Handbook of Graph Grammars*, volume 2, pages 3–61. World Scientific, 1999.

[68] J. Guadalupe Ramos, Josep Silva, and Germán Vidal. An offline partial evaluator for curry programs. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming*, WCFLP '05, pages 49–53, New York, NY, USA, 2005. ACM.

[69] Ralf Scheidhauer. *Design, Implementierung und Evaluierung einer virtuellen Maschine für Oz*. PhD thesis, Universität des Saarlandes, Fachbereich Informatik, Saarbrücken, Germany, December 1998.

[70] Zoltan Somogyi and Fergus Henderson. The design and implementation of mercury. *Joint International Conference and Symposium on Logic Programming*, Sep 1996.

[71] E Stoltz, M.P. Gerlek, and M Wolfe. Extended ssa with factored use-def chains to support optimization and parallelism. pages 43 – 52, 02 1994.

[72] Philip Wadler. How to replace failure by a list of successes. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[73] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231 – 248, 1990.

[74] David Warren. An abstact prolog instruction set. Technical report, SRI International, Aug 1983.

[75] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, April 1991.