Optimizing GHC Language Plugin Generated Code for Strict and Pure Functions via Worker/Wrapper Transformation

Justin Andresen

Master's Thesis
October 2023

Programming Languages and Compiler Construction
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Michael Hanus
M.Sc. Kai-Oliver Prott

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

GHC Language Plugins extend the capabilities of the Glasgow Haskell Compiler (GHC) by transforming code into a monadic representation that allows the usage of ambient effects in the otherwise purely functional programming language Haskell. The Curry GHC Language Plugin, for example, transforms the Haskell program into a monadic representation using a tree-based monad, effectively allowing the GHC to compile programs with Curry-like semantics. The advantage of this approach compared to the implementation of a Curry compiler from scratch is that it allows the reuse of existing language infrastructure of the GHC and a seamless integration of the transformed code into Haskell programs. However, the performance of the code generated by such plugins is suboptimal due to overheads associated with the monadic representation. This thesis introduces an optimization for GHC Language Plugins that leverages the well-known optimization technique of the worker/wrapper transformation to enhance the evaluation efficiency in particular for functions that are strict in their arguments or devoid of ambient effects. In accordance with the plugin's general philosophy, the implementation builds upon GHC's own analyses and optimizations. Our findings reveal that this optimization technique yields substantial performance enhancements, demonstrating notable reductions in both runtime execution times and memory consumption for the targeted class of functions.

Contents

1 Introduction **1**

2 Preliminaries **3**
2.1 Curry Programming Language 3
2.2 The Glasgow Haskell Compiler (GHC) 5
2.3 GHC Language Plugins 9
2.4 Traversing the AST with `syb` 15
2.5 Related Work 18

3 Optimization Approach **19**
3.1 Identification of Performance Bottlenecks 19
3.2 The Worker/Wrapper Transformation 23
3.3 Transformation Scheme 27

4 Implementation **37**
4.1 Demand Analysis 38
4.2 Effect Analysis 48
4.3 Sharing Analysis 49
4.4 Implementing the Worker/Wrapper Transformation 50
4.5 Monadic Lifting of Workers and Wrappers 57

5 Evaluation **63**
5.1 Automating Existing Tests 63
5.2 Testing Demand Signatures 66
5.3 Benchmarks 68
5.4 Maintainability 81

6 Conclusion **85**
6.1 Summary and Results 85
6.2 Limitations and Known Issues 85
6.3 Future Work 86

A Usage **89**

B Running Tests and Benchmarks **91**
B.1 Running the Test Suite 91
B.2 Running the Benchmark Suite 91
B.3 Measuring Executable Size and Compile Time 92

C Peano Numbers **95**
C.1 Common Operations 95
C.2 Tak Implementation 96

D Worker/Wrapper Transformation using unsafeBind **97**

Bibliography **99**

Chapter 1

Introduction

Curry is a functional logic programming language whose syntax and semantics are very similar to Haskell (Marlow (ed.), 2010) but extends it with features from the logic programming paradigm such as nondeterminism and free variables (Hanus (ed.), 2016). Due to the similarities of the two languages, Haskell is a reasonable target language for Curry compilers. The second version of the Kiel Curry System (KiCS2), for example, compiles Curry source code to Haskell and then uses the Glasgow Haskell Compiler (GHC) to produce machine code (Braßel, Hanus, Peemöller, and Reck, 2011). Nevertheless, a Curry compiler still has to repeat a lot of the work that has already been put into the development of Haskell compilers. For example, a Curry compiler has to implement its own type checker even though the type systems of Curry and Haskell are the same for all intents and purposes. In consequence, many useful language extensions that are available for Haskell have not been implemented for Curry yet. Besides the actual compiler, other tools like a Read-Eval-Print-Loop (REPL), a debugger, documentation generator, package manager, etc. also have to be implemented from scratch and maintained separately from their Haskell counterparts. Furthermore, interoperability with Haskell and its rich package ecosystem is not possible.

These problems motivated Prott (2020) to build a new Curry compiler based on existing infrastructure of the GHC. The *Curry GHC Language Plugin* (hereinafter referred to as the *plugin*) uses the GHC's plugin system to hook into its compilation pipeline and transforms the input Curry program into Haskell source code that can be compiled by the GHC. To implement support for Curry-specific features, the plugin performs a monadic lifting based on work by Wadler (1990) and models call-time choice semantics (Hennessy and Ashcroft, 1977) using an approach by Fischer, Kiselyov, and Shan (2011). The translation scheme of the plugin is not specific to Curry's nondeterminism. By swapping the concrete monad that is used during lifting, the plugin can be adapted to support arbitrary effects. Prott demonstrated this flexibility of the approach by creating a fork of the plugin that supports probabilism. Additional forks of the project have been created since then. The resulting class of plugins is referred to as *GHC Language Plugins*.

While the monad-based translation scheme of the plugin offers great flexibility, the resulting need to wrap and unwrap values in the monad as well as the explicit modeling of sharing introduce a significant performance overhead. In contrast to the approach chosen by KiCS2 (Brassel and Fischer, 2008), the runtime of parts of the program that are purely functional is also impaired. The primary goal of this thesis is to implement optimizations that help to improve the performance of the code generated by GHC Language Plugins. Since the performance problems are caused by the monadic representation and explicit sharing, our optimizations target functions that do not need these abstractions to work properly. This means that we predominantly focus on effect-free functions. Functions that strictly evaluate their arguments are also of particular interest to us because the monadic wrapping of their arguments is often superfluous. Since the plugin was originally conceived to minimize the

implementation effort for a new compiler by reusing parts of the GHC's infrastructure, a secondary goal of ours is to achieve these desired performance improvements by utilizing as much of the existing analyses and optimizations provided by the GHC as possible. Even though we are using the Curry GHC Language Plugin as an example throughout this thesis, our optimizations are applicable to all instances of the GHC Language Plugin.

The remainder of this thesis is structured as follows. We begin in Chapter 2 with the preliminaries of the thesis. This includes among others an introduction to the GHC's plugin system, an overview of the GHC Language Plugin's architecture and an explanation of the principles behind the monadic lifting performed by the plugin. Our approach to the optimization of the generated code is covered in Chapter 3. We continue in Chapter 4 by presenting how we implemented the formerly described optimizations. Lastly, we evaluate the performance gains achieved by our optimizations in Chapter 5 before we conclude in Chapter 6 with a summary of our work and an outlook on potential improvements and extensions to the plugin.

Chapter 2

Preliminaries

This chapter introduces fundamental concepts required to understand the rest of this thesis. Basic knowledge of the Haskell programming language (Marlow (ed.), 2010) and functional programming in general is assumed. In addition to standard Haskell 2010 language features, we also utilize language extensions provided by the GHC. While our considerations in this thesis apply to all instances of the GHC Language Plugin, we will focus on the Curry GHC Language Plugin as our primary example. Therefore, we familiarize ourselves with the Curry programming language in the first section. The second section discusses the GHC's general architecture and its plugin system in particular. How the GHC Language Plugins fit into this picture and how the plugin transforms the source code is covered in the third section. Finally, related work is outlined.

2.1 Curry Programming Language

Curry is a multi-paradigm programming language that seamlessly combines aspects from functional and logic programming (Hanus (ed.), 2016). The syntax, type-system and semantics of Curry are very similar to Haskell. The most important difference is that Curry incorporates concepts from logic programming such as nondeterminism and free variables. Of these two concepts, we will only discuss nondeterminism since free variables have the same expressive power as nondeterministic value generators (Antoy and Hanus, 2006) and are not yet supported by the plugin. Additionally, we need to understand the interplay of nondeterminism and lazy evaluation. We will cover the concept of nondeterminism in the first subsection and outline two possible interpretations of its semantics under lazy evaluation in the second subsection.

2.1.1 Nondeterminism

Provided that its evaluation terminates and throws no runtime exception, every expression in Haskell reduces to exactly one value. In this sense, Haskell expressions behave deterministically. To model a computation that can produce multiple results, the possible values need to be collected in a container such as a list or tree data structure. Curry, on the other hand, allows an expression to evaluate to zero or more values without explicitly modeling this fact on type-level. Therefore, we say that Curry possesses an *ambient* effect of nondeterminism. This is similar to how every function in Haskell can potentially fail by throwing an exception without requiring the explicit use of `Maybe` or `Either`. Haskell is therefore said to have an ambient effect of *partiality*. The partiality in Haskell is a result of incomplete pattern matches or the explicit use of built-in functions like `undefined :: a` or `error ::`

`String -> a`. Similarly, the nondeterminism of Curry arises from the use of functions that are defined in terms of *overlapping rules*. The most primitive function that possesses this property is the built-in choice operator `(?)`. On language-level, it can be defined as follows.

```
(?) :: a -> a -> a
x ? _ = x
_ ? y = y
```

The first rule evidently matches unconditionally. In Haskell, `(?)` would therefore return `x` and the second rule would never be considered. Curry, on the other hand, applies all matching rules in a nondeterministic fashion. Since the second rule also matches unconditionally, `x ? y` can be reduced to either `x` or `y`. The Curry runtime system is responsible for searching the space of possible reductions and collecting all results. If no rule matches, Curry does not trigger a runtime error like Haskell would. Instead, the computation is just irreducible, i.e., yields no results. For the purpose of creating a computation with zero results explicitly, Curry's prelude provides a primitive polymorphic operation `failed :: a` that is irreducible.

2.1.2 Run-Time Choice and Call-Time Choice

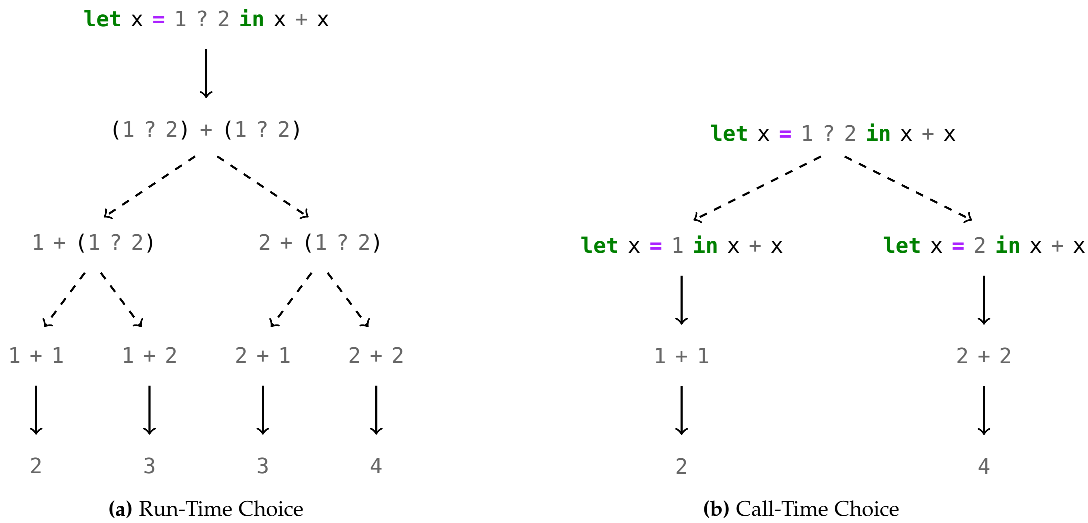
A core concept of Haskell is its *call-by-need* or *lazy* evaluation strategy. Lazy evaluation means that every expression is evaluated at most once and only when its value is needed. Consider, for example, that we perform an expensive computation *comp* and use its result in two places.

```
let x = comp in x + x
```

Due to *referential transparency*, this expression is equivalent to *comp* + *comp*. However, since *comp* is expensive, we would like to avoid evaluating it twice. Lazy evaluation ensures that `x` is only evaluated once and its value is *shared* between the two uses. While there is no way to observe that the value of an expression has been shared in Haskell, we need to be aware of interactions between sharing and nondeterminism in Curry. Consider the same example as before but instantiate the computation *comp* with a nondeterministic choice between the numbers one and two.

```
let x = 1 ? 2 in x + x
```

Due to referential transparency, we might expect that we can substitute the two occurrences of `x` with `1 ? 2` without changing the semantics of the program. As Figure 2.1a shows the expressions evaluates to the four values `2`, `3`, `3` and `4` in this situation. This interpretation of the `(?)` operator's semantics is called *run-time choice* (Hennessy and Ashcroft, 1977) as the nondeterministic choice is delayed until a concrete value for `x` is demanded. The alternative is to commit to one of the two options when `x` is bound. In this way, the concrete choice of a value for `x` is shared between the two uses. This interpretation is referred to as *call-time choice* (Hennessy and Ashcroft, 1977) and is depicted in Figure 2.1b. Using call-time choice we only get the two results `2` and `4`. These two answers coincide with the results one would expect from an eager evaluation of the program. Call-time choice is, therefore, more predictable and intuitive than run-time choice (Fischer, Kiselyov, and Shan, 2011). For this reason, the Curry programming language uses call-time choice semantics (Hanus (ed.), 2016).



(a) Run-Time Choice

(b) Call-Time Choice

Figure 2.1. Schematic evaluation using run-time choice and call-time choice. Dashed lines represent the elimination of nondeterminism, solid lines regular reduction.

2.2 The Glasgow Haskell Compiler (GHC)

The GHC is an open-source compiler and interactive environment for the Haskell programming language whose development originally started in 1989 as a research project at the University of Glasgow. Today the GHC is one of the most widely used Haskell compilers (Hudak, Hughes, Peyton Jones, and Wadler, 2007). In this section, we will give a brief overview of the GHC's architecture. We start by outlining the general compilation pipeline in the first section. Then we familiarize ourselves with the GHC's internal representation of Haskell programs as well as the intermediate language that the GHC uses for its own analyses and optimizations. Finally, the last section covers how plugins can be used to extend the GHC's functionality.

2.2.1 Compilation Pipeline

The GHC compiles Haskell programs into machine code via a series of intermediary representations. The compilation process for an input module is comprised of several phases that are executed sequentially. A simplified overview of the compilation pipeline is shown in Figure 2.2 (Marlow and Peyton-Jones, 2012). First, the *parser* reads the input file and produces an Abstract Syntax Tree (AST)

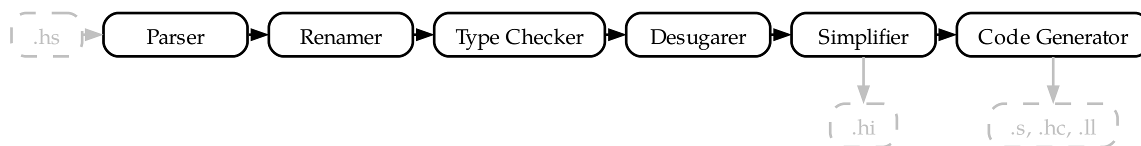


Figure 2.2. Simplified overview of the GHC's compilation pipeline.

that represents the program's source code. The *renamer* subsequently resolves all references to identifiers and type constructors before the *type checker* verifies that the program is well-typed. During both of these phases, the GHC enriches the AST with additional information about resolved entities and inferred types, respectively. Next, the *desugarer* translates the enriched AST into an intermediary language called *Core* that is easier to perform analyses and optimizations on. Depending on the selected optimization level, the GHC's *simplifier* then applies a series of syntactic transformations on the Core program such as dead code elimination, function inlining or constant folding. The Core to Core transformations are interleaved with various analyses and executed iteratively. Based on the selected backend, the *code generator* finally translates the optimized Core program into machine code, C source code or *LLVM IR*[1]. In this thesis, we are mainly concerned with the type-checked and Core representations. In the following two subsections, we will take a closer look at these two formats.

2.2.2 Abstract Syntax Tree (AST)

In addition to the Haskell 2010 standard, the GHC supports a variety of language extensions that augment Haskell's type system, introduce new syntax or provide metaprogramming capabilities. In consequence, the internal representation of the Haskell source code is quite complex. The structure of the AST is further complicated by the fact that the GHC uses a single data type to represent the parsed, renamed and type-checked program although there is information that varies between these phases such as the type of identifiers. Therefore, the AST is parameterized over a type that indicates the current phase. The GHC uses this parameter in conjunction with type families to substitute the types of phase-specific fields and constructors in the AST (Najd and Peyton Jones, 2016).

We are only concerned with the type-checked representation of Haskell programs. In this phase, identifiers have already been replaced with values that record information about the corresponding syntactic entity such as variables or data constructors. A `Var`, for example, records the name, type, scope etc. of a variable. Most importantly, these entities are identified by a `Unique` number. Their names are only used for pretty-printing at this point.

Two major kinds of AST nodes are of interest to us. Nodes of type `HsExpr` represent expressions whereas `HsBind` nodes represent both local and top-level bindings. Of these two types, there are only a few constructors that are relevant for our purposes like `HsLam`s and `FunBind`s that represent lambda abstractions and function bindings (including nullary bindings), respectively. Both of these nodes are implemented in terms of `MatchGroup`s where a match group contains multiple `Match`es each of which represents a single rule of a function definition. Additionally, we need to be aware that the AST alternates between actual nodes and special source location markers for error reporting purposes (Najd and Peyton Jones, 2016). Specific AST nodes are covered in more detail as needed throughout the thesis.

Since the AST is very complex, it is difficult to navigate. A lot of boilerplate code is needed to traverse the tree in order to update nodes or extract information. This code distracts from the actual task at hand and is prone to errors. To alleviate this problem, it is recommended (Pickering, Wu, and Németh, 2019) to use the `syb` (Scrap Your Boilerplate) library (Lämmel and Jones, 2003) to traverse the AST. We do not need to understand in detail how the library works but are going to briefly introduce the most important concepts in Section 2.4.

[1] `https://llvm.org/`

2.2.3 GHC Core

Due to the complexity of the Haskell AST, the GHC itself performs most of its own analyses and optimizations on an intermediary language called *Core* whose syntax is much simpler than that of Haskell. The Core language is based on the polymorphic lambda calculus (System F). In order to support Generalized Algebraic Datatypes (GADTs), Core extends F with type equality coercions. This extension to the polymorphic lambda calculus is known as System F_C in literature (Sulzmann, Chakravarty, Peyton Jones, and Donnelly, 2007). The Core language only consists of the following constructs:

- variables,
- literals for built-in types,
- type and value application,
- type and value lambda abstractions,
- `let` bindings without pattern matching,
- `case` expressions with flat patterns only,
- type casts via coercions and
- ticks which are used by Haskell Program Coverage (HPC)[2] to track the execution of expressions.

The central data type of the Core language's internal representation is the `CoreExpr` type. The type contains one constructor for each of the constructs listed above and, therefore, is much simpler than the Haskell AST. Variables and binders use the same `Var` type as the type-checked AST.

Besides the reduced number of syntactic constructs, the most notable difference to Haskell is that Core has explicit lambda abstractions and applications for types. Throughout the thesis, we avoid showing Core code directly. Instead, our examples make use of equivalent Haskell code. To represent Core's explicit type applications and lambdas, we are going to borrow syntax from the `TypeApplications` and `ScopedTypeVariables` language extensions.

Another big difference between Haskell and Core is that type classes are eliminated during desugaring. To implement type classes, a technique known as *dictionary passing* is used (Wadler and Blott, 1989). Dictionary passing is inspired by object oriented programming (OOP) where each object is represented in memory as a structure that contains a pointer to a class object in addition to the values of instance variables. The class object acts as a *dictionary* that maps method names to their implementations. When a method is invoked on an object, the class pointer is followed to look up the corresponding method's implementation in the dictionary (Goldberg and Robson, 1989, p. 61). In contrast to OOP, the dictionaries are not attached to individual objects but are passed as additional arguments to functions in GHC Core. There is one argument for every type class constraint that is in the function's context. Type classes are translated to a product *dictionary type* that has one field for each operation of the type class. Superclasses are represented by additional fields of the corresponding dictionary and the operations can be accessed by field selectors of the same name. For instance,

[2] http://wiki.haskell.org/Haskell_program_coverage

the following code snippet illustrates how the dictionary of the `Eq` type class is represented in Core using equivalent Haskell syntax.

```
data Eq a = Eq (a -> a -> Bool) (a -> a -> Bool)

(==) :: forall a. Eq a -> a -> a -> Bool
(==) (Eq eq _) = eq

(/=) :: forall a. Eq a -> a -> a -> Bool
(/=) (Eq _ neq) = neq
```

Type class instances are represented by top-level function bindings that construct a value of the corresponding dictionary type. The arguments of these *dictionary functions* are the dictionaries that correspond to the constraints from the instance head. The `Eq` instance for `Maybe` corresponds to the following dictionary function, for example.

```
fEqMaybe :: forall a. Eq a -> Eq (Maybe a)
fEqMaybe dEqA = Eq (eqMaybe @a dEqA) (neqMaybe @a dEqA)

eqMaybe :: forall a. Eq a -> Maybe a -> Maybe a -> Bool
eqMaybe dEqA (Just x) (Just y) = (==) @a dEqA x y
eqMaybe _    Nothing  Nothing  = True
eqMaybe _    _        _        = False

neqMaybe :: forall a. Eq a -> Maybe a -> Maybe a -> Bool
neqMaybe dEqA (Just x) (Just y) = (/=) @a dEqA x y
neqMaybe _    Nothing  Nothing  = False
neqMaybe _    _        _        = True
```

In the context of dictionary passing, the term *evidence* often refers to a variable that binds a dictionary value (or coercion). The term is used because the dictionary provides evidence that the corresponding constraint holds (or two types are equal). Evidence lambda abstractions are created for the constraints of functions. Additionally, local evidence binders are introduced by the GHC to bind dictionaries constructed by the application of a dictionary function. These local evidence binders can themselves take further evidence as arguments.

2.2.4 Compiler Plugins

As a bootstrapping compiler, the GHC is written in Haskell itself. This allows the compiler's source code to be used as a library. Additionally, the GHC can be extended through so-called *compiler plugins*. Compiler plugins are Haskell modules that are loaded by the GHC at compile time and can hook into various stages of the compilation pipeline. For instance, there are *parser plugins*, *renamer plugins* and *type checker plugins* that run after the corresponding phase has finished, i.e., operate on the original, renamed or type-checked AST respectively. Plugins can also be used to intercept loaded module

interfaces or modify Template Haskell splices. These kinds of plugins are referred to as *source plugins* as all of them operate on the GHC's internal representation of Haskell source code (Pickering, Wu, and Németh, 2019). Additionally, plugins can modify compiler flags (*driver plugin*), force module recompilation or add custom Core to Core transformations. Furthermore, the type checker can be enhanced by plugins that provide custom constraint-solving facilities. These so-called *constraint solver plugins* can be used to recover from type errors that are introduced by transformations in source plugins for example. How GHC Language Plugins make use of the GHC's plugin system will be covered in the next section.

2.3 GHC Language Plugins

GHC Language Plugins are compiler plugins that monadically lift the input program into a configurable but fixed monad. This transformation allows the program to use effects provided by the monad in an ambient fashion. For example, the Curry GHC Language Plugin is a GHC Language Plugin that is configured to use the `Nondet` monad (Prott, 2020). This monad is based on a binary tree where each leaf represents a single result of the nondeterministic computation and branches represent choices. The monad is extended by additional facilities (Fischer, Kiselyov, and Shan, 2011) to support call-time choice semantics. While the plugin's approach works with arbitrary monads and effects, we will focus on the Curry implementation for the remainder of this thesis. In this section, we cover the general architecture of the plugin and outline the monadic lifting algorithm. In the final subsection, we give an overview of the plugin's features and restrictions.

2.3.1 Architecture

In the previous section, we familiarized ourselves with the GHC's plugin system. Building on that, we will now take a closer look at how GHC Language Plugins utilize this system to implement the monadic lifting transformation. While the lifting is implemented based on the type-checked AST, each GHC Language Plugin is comprised of a driver, parser, renamer and constraint solver plugin in addition to the type checker plugin that performs the actual lifting (Prott, 2020). The following steps are performed in the individual phases.

- ▷ The **driver plugin** is responsible for suppressing the import of the Haskell prelude by enabling the `NoImplicitPrelude` language extension whereas the **parser plugin** inserts an import for a custom prelude that provides lifted versions of commonly used functions. The module used for the custom prelude is configured by the specific GHC Language Plugin and commonly compiled with the plugin itself.
- ▷ The **renamer plugin** runs after all imports have been resolved and uses this information to check whether the imported modules are lifted. If an unlifted module is encountered, the plugin aborts the compilation process and reports an error. This is necessary since the monadic lifting assumes all used functions to be lifted.
- ▷ Functions that are imported have already been lifted while the current module's source code has not been transformed prior to type checking. Therefore, a **constraint solver plugin** is employed to

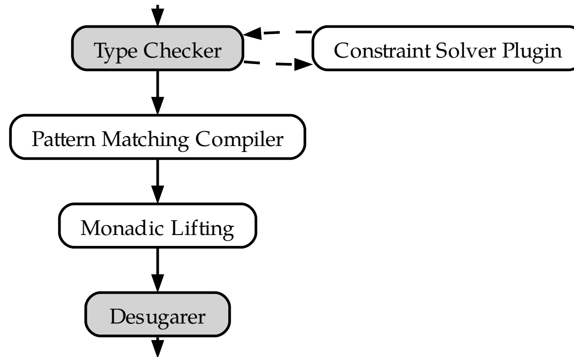


Figure 2.3. Overview of the plugin's architecture. Gray nodes are part of the regular GHC compilation pipeline. White nodes are executed as part of the plugin's translation. Dashed lines represent the flow of information while solid lines trace processing of the AST. Only phases after type checking are shown.

suppress type errors that arise due to the use of lifted functions in an unlifted context.

▷ After type checking, the **type checker plugin** performs the actual lifting as described in Section 2.3.3. To simplify lifting, the AST is preprocessed first. The most important preprocessing step is pattern matching compilation and will be covered in Section 2.3.2.

The central component of the plugin is the type checker plugin. Figure 2.3 shows how the plugin interacts with the regular GHC compilation pipeline. Pre-type-checking phases are omitted from the diagram. Neither the import mechanisms nor the constraint solving had to be adapted for the implementation of our optimizations. For the remainder of this section, we will focus on the preprocessing and lifting phases of the type checker plugin.

2.3.2 Pattern Matching Compilation

The GHC uses its Core language to simplify the compilation process. The monadic lifting on the other hand uses the type-checked AST as input. To keep the lifting simple despite the more complex representation, the plugin performs a preprocessing step that compiles nested pattern matching into flat `case` expressions and single argument lambda abstractions first. Pattern matching in guards, `do` notation and pattern bindings is also rewritten in terms of `case` expressions. The pattern matching compiler additionally desugars conditions in guards into `if` expressions. By implementing its own pattern matching compiler, the plugin can also change the semantics of pattern matching. For the most part, the algorithm by Wadler (1987) is followed. However, in contrast to Haskell, pattern matching is not performed from left to right. Instead, the plugin matches patterns in inductive positions, i.e., patterns that start with a constructor pattern in all alternatives (Hanus (ed.), 2016), first. This approach is more similar to the pattern matching semantics of Curry. Additionally, we have seen in Section 2.1 that Curry matches overlapping patterns in a nondeterministic fashion. While it would be possible for the plugin to imitate this behavior, Prott (2020) decided against it since it would limit the applicability of the plugin to Curry but the plugin is designed to work with arbitrary effects. Since the plugin also supports no free variables, the only source of nondeterminism in the program are the

built-in `(?)` and `failed` operations. After pattern matching compilation, every n-ary function binding in the program has the shape shown below on the left. Case expressions are of the form shown to the right where `pE` is a built-in function to report pattern matching failures. In case of nondeterminism, the plugin sets `pE = const failed`.

```
f = \x1 -> ... -> \xn -> e
```

```
case e of
  C1 x1^1 ... x1_k1 -> e1
  ...
  Cn x1^n ... xn_kn -> en
  _ -> pE "Non-exhaustive patterns in ..."
```

2.3.3 Monadic Lifting

Since Haskell is a purely functional programming language, there are no side effects. To express effectful computations, the effect and the order of evaluation must be modeled explicitly. While alternative approaches have been discussed in the past, the Haskell community settled on the concept of monads to model effects (Hudak, Hughes, Peyton Jones, and Wadler, 2007). A monad M is a unary type constructor for which operations with the following types can be defined such that they fulfill the *monad laws*.

```
return :: a -> M a
(>>=)  :: M a -> (a -> M b) -> M b
```

The `return` operation creates a new computation that yields the given value without any side effect. The operator `(>>=)` is known as *bind* and is used to sequence two monadic computations where the second computation can use the value returned by the first. In Haskell, these operations are provided by the `Monad` type class. To form a valid instance, the implementation of the operations must fulfill the following monad laws. Especially the left identity law will play an important role during the implementation of our optimizations.

$$\forall x, f: \quad \texttt{return}\ x \mathbin{>\!\!>\!\!=} f = f\ x \qquad \text{(Left Identity)}$$

$$\forall m: \quad m \mathbin{>\!\!>\!\!=} \texttt{return} = m \qquad \text{(Right Identity)}$$

$$\forall m, f, g: \quad m \mathbin{>\!\!>\!\!=} f \mathbin{>\!\!>\!\!=} g = m \mathbin{>\!\!>\!\!=} (\lambda x \rightarrow f\ x \mathbin{>\!\!>\!\!=} g) \qquad \text{(Associativity)}$$

To add support for ambient effects, the plugin has to rewrite the program in a monadic style. The approach followed by the plugin was first presented by Wadler (1990). The idea is to wrap the type of every value in the program whose evaluation can potentially cause an effect with the monad. In case of the Curry GHC Language Plugin, the `Nondet` monad is used. The concrete implementation of the monad is not relevant at this point. The monadic lifting works with any type constructor that implements the `Monad` type class and fulfills the monad laws.

$$[\![\tau]\!] = \begin{cases} \texttt{forall}\ \alpha\ .\ [\![\tau']\!] & \text{if } \tau = \texttt{forall}\ \alpha\ .\ \tau' \\ [\![\tau_1]\!]^r \texttt{ => } [\![\tau_2]\!] & \text{if } \tau = \tau_1 \texttt{ => } \tau_2 \\ \texttt{Nondet}\ [\![\tau]\!]^i & \text{otherwise} \end{cases}$$

(a) Regular lifting of types

$$[\![\tau]\!]^i = \begin{cases} \alpha & \text{if } \tau = \alpha \\ C^{\texttt{ND}} & \text{if } \tau = C \\ [\![\tau_1]\!]^i\ [\![\tau_2]\!]^i & \text{if } \tau = \tau_1\ \tau_2 \\ [\![\tau_1]\!]^i \texttt{ --> } [\![\tau_2]\!]^i & \text{if } \tau = \tau_1 \texttt{ -> } \tau_2 \\ [\![\tau]\!] & \text{otherwise} \end{cases}$$

(b) Inner lifting of types

Figure 2.4. Rules that govern the lifting of types. For presentational purposes, we deviate from Prott's original definition of the type lifting and actual implementation. Our set of equations is equivalent but is easier to reason about.

In the following characterization of the monadic lifting, we are going to use `Nondet` as a placeholder for the concrete monad's type constructor. Prott (2020) denotes the function that recursively lifts a type expression into the monad as $[\![\cdot]\!]$. The rules for the lifting of types are shown in Figure 2.4. Since the quantification over a type cannot introduce an effect, only the type that is quantified over must be lifted. The same is true for type class constraints. Within the constraint, type constructors must be renamed to their lifted counterparts. The recursive renaming within a type expression is denoted as $[\![\cdot]\!]^r$. Given a type constructor C, the plugin as the suffix `ND` to the lifted version, i.e., $[\![C]\!]^r = C^{\texttt{ND}}$. All other types, i.e., monomorphic types without constraints, are wrapped with the monad. Additionally, an *inner* lifting $[\![\cdot]\!]^i$ is defined that applies the lifting recursively but not at the outermost position.

One of the most important aspects for the optimizations that we are going to implement is the lifting of function types. Since the argument of a function could be an effectful computation and the application of the function can cause an effect as well, both the argument and return type must be lifted. Additionally, the function itself must be wrapped in the monad by Figure 2.4a because functions are first-class values in functional programming languages and could therefore be the result of an effectful computation. In Curry, the expression `head ? last`, for example, is a nondeterministic computation of type `[a] -> a`. The corresponding lifted type is `Nondet (Nondet [a]`$^{\texttt{ND}}$` -> Nondet a)`. To enhance readability and allow type class instances to be defined for lifted functions, the plugin defines the following `newtype` via the `TypeOperators` language extension.

```
newtype a --> b = Func (Nondet a -> Nondet b)
```

The lifting for function types as shown in Figure 2.4b uses this *lifted function arrow* type. Since the `newtype` contains the monad already, the inner lifting is sufficient for the argument and return type. The monad is wrapped around the function arrow itself by the regular lifting in Figure 2.4a.

It might also be surprising why the inner lifting is used for the lifting of type arguments in type constructor applications. Consider, for example, a list data type.

```
data List a = Nil | Cons a (List a)
```

If only the type passed to the type parameter `a` was lifted, then the list can contain effectful computations but the spine of the list would be effect-free. To allow effects in the tail of a list, all fields must

be lifted. This also means that the head of the list is lifted already. Therefore lifting the type argument would be redundant. Type synonyms and the argument of `newtype` constructor are also lifted. In contrast to `data` types, the inner lifting must be used, though (Prott, 2020). To make a seamless integration of the lifted program into Haskell possible, the plugin preserves the original type declarations and defines functions to convert between the two representations. Given a type T, these operations are provided by an instance `Normalform Nondet` T T^{ND}. The name of the type class is derived from the fact that converting to the unlifted form involves fully evaluating the value to normal form.

Finally, we need to cover how expressions are transformed by their lifting function $[\![\cdot]\!]^e$ such that for all expressions e of type τ the transformed expression $[\![e]\!]^e$ has type $[\![\tau]\!]$. Thanks to the pattern matching compiler, the plugin needs to consider only a subset of expressions. In the following, we are further restricting ourselves to the subset of expressions that are relevant for the remainder of this thesis.

▷ **Variable expressions** do not have to be changed to satisfy the invariant because the value that is bound to the variable must have been lifted already.[3]

$$[\![x]\!]^e = x$$

▷ **Lambda Abstractions** are lifted by wrapping the function in the monad and `Func` constructor. The pattern matching compiler guarantees that the only argument of the lambda abstraction is bound by a variable pattern.

$$[\![\backslash x \texttt{ -> } e]\!]^e = \texttt{return \$ Func \$ } \backslash x \texttt{ -> } [\![e]\!]^e$$

▷ **Function Applications** need to reverse the wrapping of the monad and `Func` constructor. For this purpose, the plugin defines the following helper function.

```
app :: Nondet (a --> b) -> Nondet a -> Nondet b
app mf mx = mf >>= \(Func f) -> f mx
```

A function application is then lifted by applying `app` to the lifted callee and argument.

$$[\![f\ e]\!]^e = [\![f]\!]^e \ \texttt{`app`}\ [\![e]\!]^e$$

▷ **Data constructors** do not comply with the invariant after lifting. An n-ary constructor C for a data type D has a type of the form τ_1 `->` ... `->` τ_n `->` D. After lifting, the type of C^{ND} is $[\![\tau_1]\!]$ `->` ... `->` $[\![\tau_n]\!]$ `->` D^{ND}, i.e., fields are lifted into the monad but the result (including intermediary results) are not wrapped in the monad. Therefore, the constructor needs to be wrapped with lifted lambda abstractions.

$$[\![C]\!]^e = \texttt{return \$ Func \$ } \backslash x_1 \texttt{ -> } \ldots \texttt{ return \$ Func \$ } \backslash x_n \texttt{ -> return } (C^{\text{ND}}\ x_1\ \ldots\ x_n)$$

▷ **Literals** are like nullary constructors and just need to be wrapped in the monad.

$$[\![c]\!]^e = \texttt{return}\ c$$

[3] The actual implementation is more complicated. For example, the type information attached to the identifier by the type checker needs to be updated.

▻ **Case expressions** need to unwrap the scrutinee first.

$$⟦\, \texttt{case}\ e\ \texttt{of}\ \{\ br_1, \ldots, br_n\ \}\, ⟧^e = ⟦\, e\, ⟧^e \ \texttt{>>=}\ \backslash\ x\ \texttt{->}\ \texttt{case}\ x\ \texttt{of}\ \{\ ⟦\, br_1\, ⟧^b, \ldots, ⟦\, br_n\, ⟧^b\ \}$$

Since the pattern matching compiler guarantees that there is no nested pattern matching, the branches br_i are constructor patterns whose arguments are variable patterns. They can be lifted by renaming the constructor.

$$⟦\, C\ x_1\ \ldots\ x_k\ \texttt{->}\ e\, ⟧^b = C^{\text{ND}}\ x_1\ \ldots\ x_k\ \texttt{->}\ ⟦\, e\, ⟧^e$$

The translation of `if` expressions is very similar, i.e., the condition needs to be bound first.

Given the lifting for expressions, function bindings (including local bindings) can be transformed by lifting their right-hand side. After pattern matching compilation, the plugin safely assumes that there are no arguments bound on the left-hand side and ignores them during lifting. We can exploit this fact for our optimization as we will see in Section 4.5.

The lifting of expressions and bindings as described above results in a program with run-time choice semantics. To see why this is the case consider the following lifted version of the example from Section 2.1.2.

```
let x = (?) `app' return 1 `app' return 2
 in (+ND) `app` x `app` x
```

Due to Haskell's referential transparency, this expression is indistinguishable from an expression where the occurrences of `x` are replaced by the right-hand side of the `let` expression. When $(+^{\text{ND}})$ evaluates its arguments using `(>>=)`, there is no way to know that both arguments refer to the same computation. The plugin needs to model sharing explicitly to achieve call-time choice semantics. This can be done by defining an operator `share :: Nondet a -> Nondet (Nondet a)` that converts a computation into a new computation that memorizes its result (Fischer, Kiselyov, and Shan, 2011). The `share` operator must make sure that suspended computations that are nested in the shared value are not duplicated either. This requires a traversal of the entire data structure. The recursive sharing of constructor arguments is implemented by the `shareArgs` operation from the automatically derived `Shareable` type class. In our example, the `share` operator can be used as follows to prevent repeated evaluation of `x`.

```
let x' = (?) `app' return 1 `app' return 2
 in share x' >>= \x' -> (+ND) `app` x `app` x
```

In general, all lifting rules for expressions that bind variables need to be adapted to use the `share` operator if the bound variable is used more than once on the right-hand side. Otherwise, the rules shown above still apply. For instance, the adapted lifting for `let` expressions is as follows.

$$⟦\, \texttt{let}\ x = e_1\ \texttt{in}\ e_2\, ⟧^e = \begin{cases} \texttt{let}\ x' = ⟦\, e_1\, ⟧^e\ \texttt{in}\ \texttt{share}\ x'\ \texttt{>>=}\ \backslash\ x\ \texttt{->}\ ⟦\, e_2\, ⟧^e & \text{if } x \text{ occurs more than once in } e_2 \\ \texttt{let}\ x' = ⟦\, e_1\, ⟧^e\ \texttt{in}\ ⟦\, e_2\, ⟧^e & \text{otherwise} \end{cases}$$

2.3.4 Limitations

As has already been mentioned, the plugin lacks support for free variables and overlapping rules. Another Curry feature that is not supported by the plugin is functional patterns (Antoy and Hanus, 2005). While these restrictions are specific to the Curry GHC Language Plugin, there are also limitations that are inherent to the monadic transformation.

Effects in recursive local bindings are not properly shared. In the translation rule for `let`, we can see that the `share` operator only scopes over e_2. Therefore, recursive occurrences of the local variable inside e_1 are not shared. This can change the semantics of the program as the following example demonstrates.

```
let bits = (0 ? 1) : bits in take 3 bits
```

When this expression is evaluated with KiCS2, the only two results are `[0, 0, 0]` and `[1, 1, 1]` as one would expect from call-time choice semantics. The plugin, on the other hand, produces one list for each of the seven possible combinations three bits can be arranged in. However, in practice, this issue has only little relevance.

The most severe issue is that the translation of type classes is broken in the current version of the plugin. In particular, it is not possible to translate an instance for a custom type class that has just one operation. For example, the following code leads to a compile-time error.

```
class Foo a where
  foo :: a -> a

instance Foo Int where
  foo = (+1)
```

The reason for the crash is most likely related to the fact that the GHC uses a `newtype` declaration for the dictionary of type classes with a single operation. Defining instances of type classes with multiple operations or built-in type classes works as expected. For example, replacing `Foo` with the following definition fixes the previous example.

```
class Foo a where
  foo :: a -> a
  bar :: a -> ()
  bar = const ()
```

2.4 Traversing the AST with syb

As we have noticed in Section 2.2.2, the GHC's AST is very complex. Therefore, the plugin employs the `syb` library (Lämmel and Jones, 2003) to traverse the AST. The `syb` library leverages generic programming techniques to provide a uniform interface for updating and querying values of a specific

type that are embedded at arbitrary depth within some (potentially mutually recursive) data structure. The `Typeable` and `Data` type classes are used to implement this functionality. The `Typeable` type class provides facilities for type-safe casts. The `Data` type class provides primitives that can be used to map over the immediate subterms of a term. Using the `DeriveDataTypeable` language extension, both of these classes can be derived automatically. Therefore, we do not have to be concerned with any implementation details. In this section, we cover the most important concepts of the `syb` library.

2.4.1 Generic Transformations

First, we need to understand the concept of *type extensions* and *generic transformations*. Given some fixed type A and a function f `::` A `->` A, the type extension of f is a polymorphic function g `:: b -> b` that is characterized by the following equation.

$$g\ x = \begin{cases} f\ x & \text{if } \texttt{x :: A} \\ x & \text{otherwise} \end{cases}$$

The type extension g can be applied to arbitrary values and leaves them unchanged unless they are of type A, in which case they are transformed using f. A function like g is called a *generic transformation* because it can be applied to values of any type and maps them to values of the same type. The `syb` library provides a function `mkT` (read "make transformation") that converts a function f to its type extension g, i.e., `mkT` $f = g$ To tell whether the argument of g is of type A, `mkT` uses a type-safe cast operation provided by the `Typeable` type class.

```
mkT :: (Typeable a, Typeable b) => (a -> a) -> b -> b
```

While the resulting generic transformation can be applied to arbitrary values, we actually want to apply the function at arbitrary depth within a data structure. In a first approximation, we can use the following operation from the `Data` type class to apply the generic transformation to all immediate subterms of a value of type `a`. To allow the argument of the function to be polymorphic, the `Rank2Types` language extension is required.

```
gmapT :: (forall b. Data b => b -> b) -> a -> a
```

Building on this primitive, the `syb` library provides a function `everywhere` that recursively applies a transformation to all immediate subterms of a value. The type of `everywhere` is shown below. Both the argument and return type are generic transformations.

```
everywhere ::
  (forall a. (Data a) => a -> a) ->
  (forall a. (Data a) => a -> a)
```

There are also `mkM` and `everywhereM` which are monadic variants of `mkT` and `everywhere`. Since the order in which the nodes are visited is relevant in case of monadic transformations, it is important to note that `everywhereM` performs a bottom-up traversal.

2.4.2 Generic Queries

While generic transformations allow us to update AST nodes at arbitrary depth, *generic queries* are used to extract information from the AST and aggregate the results. Again we start with a concrete function `f :: A -> R` that extracts a value of type R from an AST node of type A. We want to extend this function to a polymorphic `g :: b -> R` that can be applied to values of any type. This time, we additionally, need a value `z :: R` that we can fallback to in case the argument is not of type A.

$$g\;x = \begin{cases} f\;x & \text{if } \texttt{x :: A} \\ z & \text{otherwise} \end{cases}$$

To construct g for a given f and z, we can use the `mkQ` function provided by `syb` where `mkQ` $z\;f = g$. To apply the resulting generic query at arbitrary depth, the `everything` combinator is used. Since the data structure can contain multiple values of type R, we need to provide a function that combines them. The types of `mkQ` and `everything` are shown below.[4].

```
mkQ :: (Typeable a, Typeable b) => r -> (a -> r) -> b -> r
everything :: forall r.
  (r -> r -> r) ->
  (forall a. (Data a) => a -> r) ->
  (forall a. (Data a) => a -> r)
```

There is a variation of `everything` that can stop the traversal early if a certain condition is met. The given generic query simply returns an additional flag that indicates whether the traversal should continue or not.

```
everythingBut :: forall r.
  (r -> r -> r) ->
  (forall a. (Data a) => a -> (r, Bool)) ->
  (forall a. (Data a) => a -> r)
```

A common use-case for generic queries is to collect all occurrences of a certain type of AST node that satisfy a given predicate. The `listify` function provided by `syb` implements this functionality.

```
listify :: (Typeable r) => (r -> Bool) -> GenericQ [r]
```

There are many more forms of generic traversals provided by the `syb` library. We will introduce these functions as needed in the following chapters.

[4] For presentational purposes, we have expanded the type synonym `GenericQ`.

2.5 Related Work

Since its initial release by Prott in 2020, the plugin has undergone continued development. While a significant portion of this effort involved adapting the plugin to newer versions of the GHC, there have also been notable contributions from various theses and papers that have expanded upon the plugin's functionality. In this section, we provide a brief overview of key advancements and contributions made in these publications. Furthermore, we discuss other work related to the topic of this thesis.

In their bachelor thesis Wieczerkowski (2021) implemented a fully-featured Curry system based on the Curry GHC Language Plugin. Their Curry system adds support for language features like free variables, unification, Input/Output (IO) and encapsulated search that were missing from the original plugin. Additionally, they implemented a REPL as an interface for the Curry system. Large parts of their work are implemented at language-level with only minor changes to the plugin itself. While these additions have not yet been incorporated into the plugin's code base, the design of their system should make an integration with our optimizations trivial.

That the plugin is not only useful in context of the Curry programming language, has been demonstrated by Teegen, Prott, and Bunkenburg (2021). Their *inversion plugin* is based on the GHC Language Plugin and allows to automatically invert functions by extending Haskell's computational model by nondeterminism and free variables. The inverse functions are used by the inversion plugin to add support for *functional patterns* (Antoy and Hanus, 2005) to Haskell.

There have also been attempts to improve the performance of the plugin's generated code in the past. In their bachelor thesis, Lange (2022) explored the feasibility of using a technique known as *fold/build fusion* or *Short Cut Deforestation* (Gill, Launchbury, and Peyton Jones, 1993) to eliminate binds from the generated code. Similar to our approach, their work also heavily relies on the GHC's rewriting mechanism (Peyton Jones, Tolmach, and Hoare, 2001) to achieve the optimization. Unfortunately, their evaluation did not show any significant performance improvements even for constructed examples. On the contrary, they report a 50 % reduction in runtime performance and advise against using fold/build fusion in the context of the plugin.

A more promising outlook is provided by recent research of Hanus, Prott, and Teegen (2022) on a more efficient monadic representation for nondeterministic computations. Their monad is a purely functional implementation of *memoized pull-tabbing* (Hanus and Teegen, 2020). Since their monad is able to prevent repeated sharing of the same value, it can avoid large parts of the performance problems that we will encounter in Section 3.1. They conducted benchmarks that indicate good performance compared to other Curry implementations.

The main idea of this thesis is to transform the program based on a demand analysis to improve the evaluation of strict arguments. That demand analysis can be used to improve the performance of lazy nondeterministic computations by evaluating demanded arguments in a strict manner has previously been demonstrated by Hanus (2012).

Chapter 3

Optimization Approach

This chapter covers the approach that we have taken to enhance the performance of the code generated by the plugin. The first section investigates the underlying factors contributing to the performance problems currently exhibited by the generated code. Subsequently, we introduce the general concept of a commonly used optimization technique known as the worker/wrapper transformation. Lastly, we present how we can employ the worker/wrapper transformation to optimize the code generated by the plugin.

3.1 Identification of Performance Bottlenecks

In this section, we investigate the performance problems of the code generated by the plugin. To this end, we analyze the runtime behavior of two example programs. We start by considering a naive implementation of the factorial function as shown below.

```
fac_Int :: Int -> Int
fac_Int n = if n <= 0 then 1 else n * fac_Int (n - 1)
```

Additionally, we want to consider a variation of this function based on Peano numbers instead of the built-in `Int` type. Peano numbers are an inductively defined representation of natural numbers based on their axiomatic characterization by Giuseppe Peano in 1889. To represent them in Haskell, we will be using the following algebraic data type.

```
data Nat = O | S Nat
```

The constructor `O` represents the number $0 \in \mathbb{N}$ and an application of the constructor `S` yields the successor $n + 1$ for any given $n \in \mathbb{N}$. Intuitively speaking, any natural number n is representable as a chain of n applications of the constructor `S` terminated by a final `O`.

$$\forall n \in \mathbb{N} : n \equiv \underbrace{\texttt{S (S ... (S}}_{n-\text{times}}\texttt{ O) ...)}$$

Before we can define the Peano-based factorial function, we need to define operations for the addition and multiplication of two Peano numbers. These operations are omitted at this point for brevity but can be found in Appendix C for reference.

```
facNat :: Nat -> Nat
facNat O       = S O                   -- 0! = 1
facNat m@(S n) = mult m (facNat n)    -- ∀n ∈ ℕ: (n + 1)! = (n + 1) · n!
```

The runtime of the compiled programs as well as the corresponding lifted versions is shown in Figure 3.1 for different values of n. Note the logarithmic scale of the y-axis. The integer-based implementation has linear runtime as one would expect from the factorial function. The lifted variant runs about 100 times slower than the original function but is still linear in nature. For the Peano-based implementation, on the other hand, no polynomial complexity can be expected. The runtime must possess a lower bound of $n!$ since at least that many applications of the `S` constructors have to be chained to constitute the result. Therefore, it is not surprising to see the plotted runtime of the Peano-based implementation grow rapidly. However, after lifting, the rate of growth accelerates sharply. The final run for the lifted version of the Peano-based implementation even had to be excluded because the function exhausted all available memory before completing.

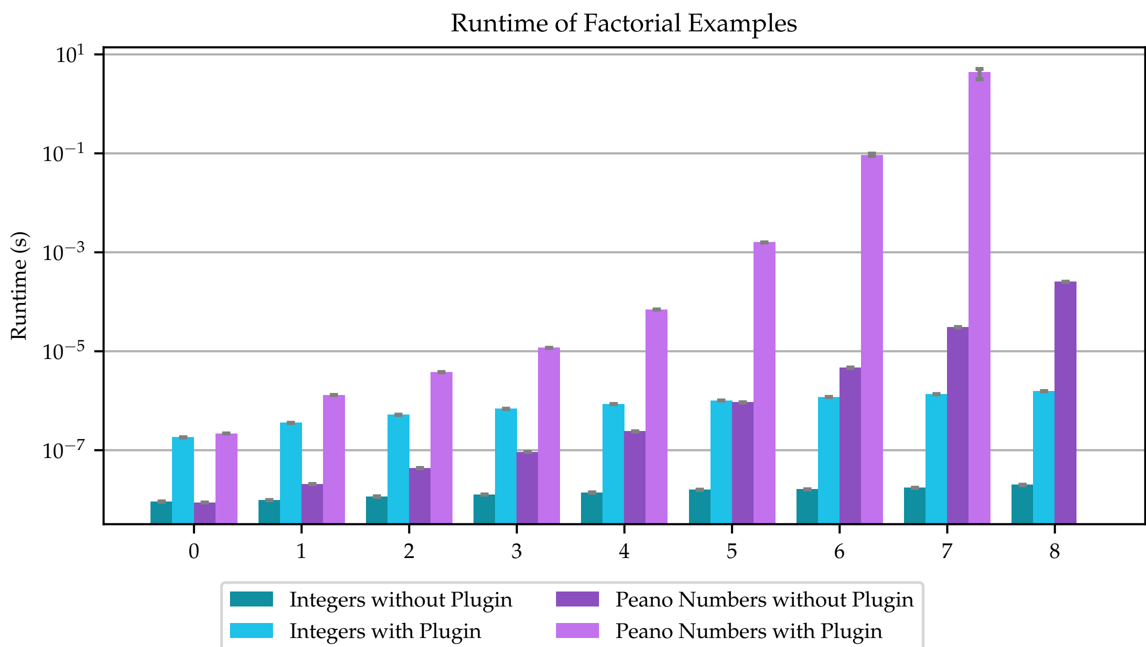


Figure 3.1. Runtime measured for the two implementations of the factorial functions for values of n between 0 and 8. See also Section 5.3 for details about our benchmarking methodology.

To deepen our understanding, let us consider what the lifted versions of the two functions look like. Below is a simplified version of the code generated by the plugin for the Peano-based implementation.

```
facNat^ND :: Nondet (Nat --> Nat)
facNat^ND =
  return $ Func $ \mn ->
    share mn >>= \sn ->
```

```
sn >>= \n -> case n of
  O -> return (S (return O))
  S mn' -> multND `app` sn `app` (facNDNat `app` mn')
```

The entire function is wrapped in a `return` because as first-class citizens of the language functions can themselves be the result of an effectful computation. This fact is also reflected in the type signature. The `Func` constructor right after the `return` is just needed to create an instance of the lifted function arrow (`-->`). In the body of the function, the monadically lifted parameter `mn` is shared using the `share` operator because the parameter is used twice on the right-hand side: once during pattern matching and once in the multiplication. Immediately afterwards, the shared value `sn` is taken out of the monad in order to perform the pattern match. In the first branch of the `case` expression, the constructor applications are `return`ed in order to lift them into the monad[1]. In the second branch, the `app` operator is used to apply the multiplication and factorial functions. The functions cannot be applied directly, because they are lifted into the monad and wrapped by the `Func` constructor.

The lifting of the integer-based implementation is conceptually similar but syntactically more contrived because more function applications are involved.

```
facNDInt :: Nondet (Nat --> Nat)
facNDInt =
  return $ Func $ \mn ->
    share mn >>= \sn ->
    ((<=ND) `app' sn `app` return 0) >>= \isLess >>=
      if isLess
        then return 1
        else (*ND) `app` sn `app` (facNDInt `app` ((-ND) `app` sn `app` (return 1)))
```

Comparing the original definitions with their lifted counterparts, we identify the following three major aspects that could affect performance.

1. **The need to bind functions using the `app` operator in order to apply the function.** As stated above, the `return` and `Func` are needed in case the function is the result of an effectful computation. However, in our example, this is not the case since all involved functions are pure. In general, all n-ary function bindings can be (partially) applied to n arguments or less without causing an ambient effect. In these cases, we would like to avoid the indirection through the monad and apply the underlying function directly.

2. **The bind operator `>>=` that needs to be inserted whenever a `case` or `if` expression is used.** These operations are redundant when the scrutinee or condition has just been `return`ed. For example, `(-ND)` is strict in its second argument and therefore immediately binds the result of the argument `return 1`. Additionally, the result of the subtraction is wrapped in the monad but in the recursive call of $\texttt{fac}^{\texttt{ND}}_{\texttt{Int}}$ the result is immediately unwrapped again because the factorial function is also strict in its argument. Therefore, we would like the argument of strict functions not to be wrapped in the

[1] For presentational purposes, this part has been greatly simplified. As has been presented in Section 2.3.3, constructor applications would usually be translated as an immediately applied lifted lambda abstraction. The GHC is already capable of eliminating the lambda abstraction, though.

monad, i.e., we want to use call-by-value semantics for strict arguments. In case a monadic value is passed to the function, the calling function should be responsible for unwrapping the argument.

3. **The explicit sharing of the argument using the `share` operator.** The performance penalty is especially high in case of Peano numbers since the argument is a value of a recursive data structure and `share` traverses the structure using `shareArgs`. Furthermore, in the next recursive call, the entire data structure will be shared again. In order to optimize performance we should strive to minimize the number of `share` operations.

To assess the extent to which the individual aspects contribute to the observed performance problems, we next conduct a series of micro-benchmarks. To this end, we define the following three classes of functions that test the influence of `app`, `(>>=)` and `share` in isolation.

▷ The first class of functions applies the `id` function n times to the argument such that we end up with n applications of the `app` operator in the lifted version.[2]

```
applyₙ x = id (id (... (id x)))
```

▷ For the second benchmark, we perform an n levels deep pattern match on the argument, i.e., the lifted equivalent contains n nested binds.

```
matchₙ ((((x, _), ...), _), _) = x
```

▷ Finally, we benchmark a class of functions that duplicate their argument n times using `let` bindings and then use each of the n local variables twice to force a `share` to be inserted during lifting.

```
shareₙ x₀ = let { x₁ = x₀, x₂ = x₁, ..., xₙ = xₙ₋₁} in [ (x₁, x₁), (x₂, x₂), ..., (xₙ, xₙ) ]
```

Figure 3.2 shows the runtime of these functions and their lifted counterparts for varying values of the meta variable n between 0 and 10. In all cases, we just passed the unit value `()` as an argument to the functions. We clearly see that the lifted versions are consistently slower than their unlifted counterparts. Across all functions, there is a difference of about one order of magnitude. Furthermore, the runtime of all functions is linearly correlated with the value of n even though the logarithmic scale makes this fact not immediately obvious. While at 22ns the slope for the $\texttt{apply}_n$ and $\texttt{match}_n$ functions is similar, the $\texttt{share}_n$ functions grows 10 times faster at 270ns per unit increase. Since we only considered primitive values, the benchmark does not yet capture deep sharing behavior. Thus, the performance impact of `share` is even higher in practice.

[2] To prevent the GHC from distorting the result by applying premature optimizations, we added the `NOINLINE` pragma to our implementation of the `id` function.

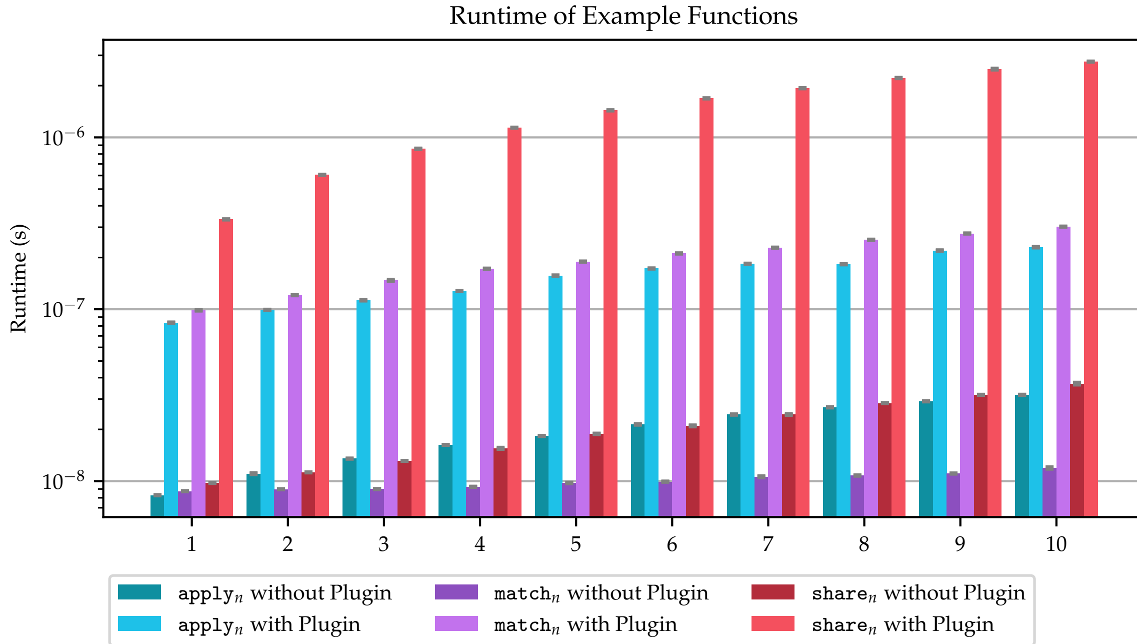


Figure 3.2. Results of the runtime measurements for `apply`$_n$, `match`$_n$ and `share`$_n$ for different values of the meta variable n between 0 and 10 before and after lifting.

In conclusion, all three aspects have a measurable impact on the performance of the generated code. However, we believe that the deep sharing mechanism acts as the primary bottleneck regarding runtime due to its recursive nature. Even though all parts of the generated code are relevant to uphold the plugin's invariants, the key to optimizing the generated code is to find conditions under which we are allowed to deviate from the general translation scheme. In the following two sections, we will introduce the worker/wrapper transformation as our primary tool in this endeavor.

3.2 The Worker/Wrapper Transformation

In this section, we present the general concept of a commonly used optimization technique known as the worker/wrapper transformation. While the technique has been used in the field of compiler construction long before (Peyton Jones and Launchbury, 1998), the first formal description of the transformation was given by Gill and Hutton (2009). The goal of the transformation is to change the type of a function in a way that makes it more amenable to optimization without affecting other functions that depend on it. To this end, the function is split into two new functions.

- ▻ The *worker* function is the part that actually performs the computation. The implementation and type are changed as part of some optimization.
- ▻ The *wrapper* function, on the other hand, has the original type and name of the function and is solely responsible for correctly calling the worker function.

As a result, the changed calling convention of the worker does not have to be respected at call-sites since the wrapper function takes care of interfacing with the worker while exposing the original function's interface itself. By inlining the wrapper function, the compiler can then apply further optimizations, ideally eliminating all of the interfacing logic and leaving only a call to the more efficient worker function (Sergey, Peyton Jones, and Vytiniotis, 2017, p. 5).

The worker/wrapper transformation has not just been experimentally shown to yield performance benefits but can even be formally verified to to either preserve or improve runtime performance under certain conditions (Hackett and Hutton, 2014). While we attempt to formally convince ourselves of the correctness of our work in this thesis, runtime improvements are solely assessed based on benchmarks in Chapter 5.

3.2.1 Formal Definition

Now that we have outlined the basic idea of the worker/wrapper transformation, let us take a closer look at the formal definition. We will be following the framework laid out by Gill and Hutton (2009).

First, consider a potentially recursive computation $comp$ of some type A. The computation can be defined in terms of the fixpoint combinator (Peyton Jones, 1987, p. 26), i.e., the computation satisfies the equation below for some $body$ `::` A `->` A.

$$comp = \text{fix}\, body$$

Next, let $wrap$ `::` B `->` A and $unwrap$ `::` A `->` B be functions that convert between the original type A and some other type B. Specifically, $unwrap$ transforms the original computation into the new type and $wrap$ reverses this transformation. These functions do not necessarily need to be implemented in Haskell. They can also be meta functions that rewrite the source code syntactically. Most importantly, though, they need to satisfy one of the following so-called *worker/wrapper assumptions* where id_A `::` A `->` A denotes the identity on A.

$$wrap \circ unwrap = id_A \tag{WW1}$$
$$wrap \circ unwrap \circ body = body \tag{WW2}$$
$$\text{fix}(wrap \circ unwrap \circ body) = \text{fix}\, body \tag{WW3}$$

All of these three criteria basically state that the two functions are inverses of each other. While the first equation enforces this relationship to hold in general, the two other equations only require it to hold in the context of the specific computation that we want to transform.

Given these building blocks, we can define how a binding for a function `f` that computes $comp$ is split into a corresponding worker and wrapper function as depicted in Figure 3.3. On the left side, the original function binding instantiates the $body$ of the computation with `f` since the function can call itself recursively. On the right side, it would also be correct for the worker function to use the wrapper `f` for recursion but instead the wrapper's right-hand side $wrap$ $\texttt{f}_{\texttt{worker}}$ is inlined into the $body$ of the worker to break the mutual recursion between the two resulting functions.

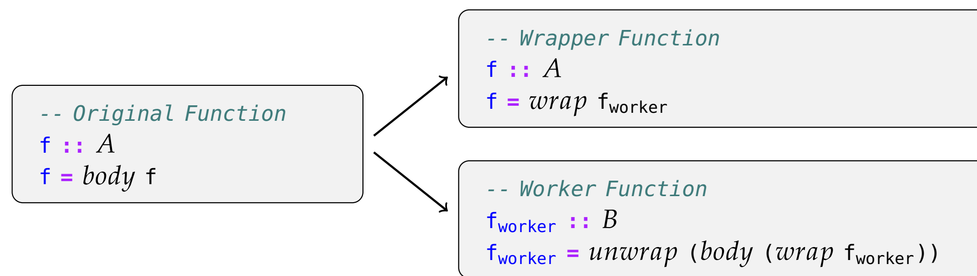


Figure 3.3. Schematic depiction of the worker/wrapper transformation. On the left is the original binding for the function `f`. On the right are the resulting bindings after worker/wrapper transformation. The binding on the top is the wrapper function with the same name as the original. On the bottom, the binding of the worker function $\texttt{f}_{\texttt{worker}}$ is shown.

3.2.2 Examples

Before we apply the worker/wrapper transformation scheme to our setting, we will first take a look at examples of how the transformation is used in practice.

One of the very first documented instances of the worker/wrapper transformation concerned the optimization of the factorial function (Peyton Jones and Launchbury, 1998). Even though we just covered the factorial function in Section 3.1, let us first recapitulate the definition of the integer-based implementation.

```
facInt :: Int -> Int
facInt n = if n <= 0 then 1 else n * facInt (n - 1)
```

In a non-strict language like Haskell, variables are only evaluated when they are used. Thus, a variable of type `Int` is not represented by a native integer but rather by a pointer to a *box* somewhere on the heap. The box can either contain the underlying *unboxed* integer or – in case it has not been evaluated yet – a so-called *thunk* that computes the actual value when it is needed. This indirection is necessary to support lazy evaluation. However, it also comes with a performance penalty since the indirection needs to be resolved at runtime. For example, in the recursive call of the factorial function, first, a new box is allocated and a thunk for decrementing `n` is stored inside. However, right as we enter the recursive call, the parameter is immediately demanded by the `(<=)` operator. Thus, the thunk is taken out of the box and evaluates `n - 1` which itself involves multiple steps of boxing and unboxing values.

Peyton Jones and Launchbury (1998) solved the problem of repeated boxing and unboxing by making unboxed integers first-class citizens of the language. This is achieved by introducing a new type `Int#` that represents unboxed integers. The boxed `Int` type can then be defined as a wrapper around the unboxed version.

```
data Int = I# Int#
```

The `Int#` type along with corresponding unboxed versions of the arithmetic operations `(<=#)`, `(*#)` and `(-#)` can then be used to define a new version of the factorial function that does not involve any boxing or unboxing.

```
fac_Int# :: Int# -> Int#
fac_Int# n = if n <=# 0 then 1 else n *# fac_Int# (n -# 1)
```

This function is incompatible with the original factorial function since the types of the parameters and return values differ. Thus, the following wrapper function is introduced that simply unboxes the given integer, calls the worker function and boxes the result again.

```
fac_Int :: Int -> Int
fac_Int (Int n) = I# (fac_Int# n)
```

The data type `Int#` is exposed to the user via GHC's `MagicHash` language extension. However, users usually do not have to deal with the unboxed versions of primitive data types or worker/wrapper transform functions themselves. This is because the GHC actually performs a worker/wrapper transformation as part of its regular compilation pipeline (Sergey, Peyton Jones, and Vytiniotis, 2017). This worker/wrapper transformation is not limited to unboxing of primitive data types. Consider, for example, the following function that takes a pair of type (A, B) as an argument and performs an expensive computation $comp$ `::` A `->` C that only involves the first component of the pair.

```
f :: (A, B) -> C
f p = case p of
  (x, _) -> comp x
```

The GHC analyzes the program and determines that:

1. the function is strict in its first argument and
2. the second component is absent from the right-hand side.

Therefore, the GHC can generate a worker function that only takes the first component of the pair as an argument and moves the pattern matching to a wrapper function.

```
-- Wrapper Function
f :: (A, B) -> C
f p = case p of
  (x, _) -> f_worker x

-- Worker Function
f_worker :: A -> C
f_worker x =
  let p = (x, error "absent")
   in comp x
```

In case the computation also depends on the removed parameter `p`, the GHC introduces a local binding that reconstructs the pair. The absent field can be ⊥ as the analysis already concluded that the field is never forced. In our example, the *comp* function does not depend on `p`. In consequence, the GHC will remove the binding in a later step of the compilation pipeline.

3.3 Transformation Scheme

In the previous section, we familiarized ourselves with the concept of the worker/wrapper transformation and have seen an example of how the GHC uses it internally to unbox arguments of strict functions among others. In this section, we will outline how we can employ this transformation to optimize code generated by the plugin. In the first three subsections, we are going to tackle one of the bottlenecks that we identified in Section 3.1 each. In the fourth subsection, we discuss an extension to our worker/wrapper transformation that yields additional performance improvements but we have not been able to fully implement. We conclude the presentation of the ideas behind our worker/wrapper transformation in the fifth subsection by summarizing the general transformation scheme.

3.3.1 Unlifting Functions

The first aspect that we identified in Section 3.1 is that an invocation of a monadically lifted function of type `Nondet (A --> B)` via the `app` operator is much slower than a direct invocation of a function `Nondet A -> Nondet B` that is neither wrapped by the monad nor the constructor `Func` of the lifted function arrow (`-->`). As we have learned in the previous section, the worker/wrapper transformation is a good candidate when you want to change the type of a computation. First, we need to define a mapping $unwrap_{\mathsf{Func}}$ that transforms a function of type `Nondet (A --> B)` to the target type `Nondet A -> Nondet B`. The function is characterized by the equation below.

```
unwrap_Func :: Nondet (A --> B) -> Nondet A -> Nondet B
unwrap_Func (return (Func f)) = f
```

This definition captures the syntactic removal of the monadic wrapper and the constructor `Func` that would usually be generated around a lambda abstraction. Note that the domain is limited to computations that resulted directly from the lifting of a lambda abstraction. In the inverse direction, we can define a mapping $wrap_{\mathsf{Func}}$ that reintroduces the monadic wrapper and the constructor `Func` to restore the original type of the computation.

```
wrap_Func :: (Nondet A -> Nondet B) -> Nondet (A --> B)
wrap_Func f = return (Func f)
```

In order for a worker/wrapper transformation using these two functions to be valid, they need to satisfy at least one of the worker/wrapper assumptions. Therefore, we prove that $wrap_{\mathsf{Func}}$ and $unwrap_{\mathsf{Func}}$ satisfy the second worker/wrapper assumption (WW2) under the condition that the

transformed function has a lifted lambda abstraction on its right-hand side. The more general worker/wrapper assumption does not hold due to the need for this contextual clue.

Proof. Let A and B be arbitrary types and consider a function that an be expressed as the least fixed point of a function *body* defined as shown below for some *body'* `:: Nondet (A --> B) -> Nondet A -> Nondet B`.

```
body  :: Nondet (A --> B) -> Nondet (A --> B)
body f = return (Func (\mx -> body' f mx))
```

Then the following equation holds for all f `:: Nondet (A --> B)`.

$$
\begin{aligned}
& wrap_{\texttt{Func}}\,(unwrap_{\texttt{Func}}\,(body\,f)) \\
&= wrap_{\texttt{Func}}\,(unwrap_{\texttt{Func}}\,(\texttt{return (Func (\textbackslash mx-> } body'\,f\texttt{ mx)))})) && \text{(Def. } body\text{)} \\
&= wrap_{\texttt{Func}}\,(body'\,f) && \text{(Def. } unwrap_{\texttt{Func}}\text{)} \\
&= \texttt{return (Func (\textbackslash mx-> } body'\,f\texttt{ mx))} && \text{(Def. } wrap_{\texttt{Func}}\text{)} \\
&= body\,f && \text{(Def. } body\text{)}
\end{aligned}
$$

□

To demonstrate how this transformation works in practice, let us consider the lifting of the identity function as shown below on the left. After worker/wrapper transformation with $unwrap_{\texttt{Func}}$ and $wrap_{\texttt{Func}}$, we obtain the worker and wrapper functions characterized by the equations on the right.

```
id^ND :: Nondet (a --> a)
id^ND = return (Func (\mx -> mx))
```

```
id^ND :: Nondet (a --> a)
id^ND = wrap_Func id^ND_worker
      = return (Func (\mx -> id^ND_worker mx))

id^ND_worker :: Nondet a -> Nondet a
id^ND_worker = unwrap_Func (return (Func (\mx -> mx)))
             = \mx -> mx
```

3.3.2 Unlifting Strict Arguments

While the previously discussed transformation elides the monad around the function arrow, the function's argument is still wrapped in the monad after the transformation. The monad is needed because a caller might pass an effectful computation to the function. However, as we have seen in Section 3.1 already, the need to wrap a value in the monad in cases where the argument will be bound by the function right away can degrade performance. Therefore, we want to follow the example of

the GHC and elide the monad around the argument where possible similar to how the GHC unboxes strict arguments.

This time we have to convert functions of type `Nondet` A `->` `Nondet` B to functions of type A `->` `Nondet` B. The function $unwrap_{\texttt{Strict}}$ defined below does exactly that by lifting the argument into the monad before calling the original function. Conceptually, this operation corresponds to syntactically replacing all occurrences of the original argument `mx` with `return x`.

```
unwrapStrict :: (Nondet A -> Nondet B) -> A -> Nondet B
unwrapStrict f x = f (return x)
```

The other direction just involves binding the argument before applying the function. Therefore, we can define $wrap_{\texttt{Strict}}$ as shown below.

```
wrapStrict :: (A -> Nondet B) -> Nondet A -> Nondet B
wrapStrict f mx = mx >>= \x -> f x
```

Again, we would like to verify that this kind of transformation is allowed by proving that one of the worker/wrapper assumptions is satisfied. Since we have restricted ourselves to strict functions, we suppose that this information must play a role in the proof. Therefore, we can rule out the most general worker/wrapper assumption (WW1) and instead attempt a proof of the second assumption (WW2) under the precondition that the least fixed point of *body* `:: (Nondet` A `-> Nondet` B`) -> Nondet` A `-> Nondet` B is strictly evaluating its argument at some point in the returned computation. Because this concept is difficult to capture formally, we first consider a weaker condition where the only bind of the argument is the outermost operation in the *body*. To this end, we define *body* in terms of a helper function *body'* `:: (Nondet` A `-> Nondet` B`) ->` A `-> Nondet` B as shown below.

```
body f mx = mx >>= \x -> body' f x
```

The second worker/wrapper assumption is then satisfied as the following equation demonstrates given an arbitrary f `:: Nondet` A `-> Nondet` B.

$$
\begin{aligned}
& wrap_{\texttt{Strict}}\,(unwrap_{\texttt{Strict}}\,(body\,f)) \\
&= wrap_{\texttt{Strict}}\,(unwrap_{\texttt{Strict}}\,(\texttt{\textbackslash mx-> mx >>= \textbackslash x->}\ body'\,f\ \texttt{x})) && \text{(Def. } body\text{)} \\
&= wrap_{\texttt{Strict}}\,(\texttt{\textbackslash y-> return y >>= \textbackslash x->}\ body'\,f\ \texttt{x}) && \text{(Def. } unwrap_{\texttt{Strict}}\text{)} \\
&= wrap_{\texttt{Strict}}\,(\texttt{\textbackslash y->}\ body'\,f\ \texttt{y}) && \text{(Monad Left Identity)} \\
&= \texttt{\textbackslash mx-> mx >>= \textbackslash x-> (\textbackslash y->}\ body'\,f\ \texttt{y) x} && \text{(Def. } wrap_{\texttt{Strict}}\text{)} \\
&= \texttt{\textbackslash mx-> mx >>= \textbackslash x->}\ body'\,f\ \texttt{x} && (\beta\text{-reduction}) \\
&= body\,f && \text{(Def. } body\text{)}
\end{aligned}
$$

In general, the bind is not the outermost operation, though. Thus, we have to relax the precondition on *body*. However, if we attempt the proof, we eventually get stuck. The problem prevails even if we use

the least general worker/wrapper assumption (WW3) instead. The reason is that the worker/wrapper assumptions are violated by the transformation. To see why this is the case, consider the following counterexample.

```
c :: Nondet A -> Nondet B
c mx = my >>= \y -> mx >>= \x -> c' x y
```

Before this function evaluates its argument `mx`, it first runs another computation `my`. After applying the worker/wrapper transformation, we obtain the following worker and wrapper functions.

```
c_wrapper :: Nondet A -> Nondet B
c_wrapper mx = mx >>= \x -> c_worker x

c_worker :: A -> Nondet B
c_worker x = my >>= \y -> (return x) >>= \x -> c' x y
```

If we now consider an invocation of the wrapper function $c_{wrapper}$ to an arbitrary computation mx `:: Nondet` A, we notice that the order of binds has been reversed compared to the original function `c`.

```
c_wrapper mx = mx >>= \x-> c_worker x
             = mx >>= \x-> my >>= \y-> (return x) >>= \x-> c' x y
             = mx >>= \x-> my >>= \y-> c' x y
             ≠ my >>= \y-> mx >>= \x-> c' x y
             = c mx
```

The unboxing of strict arguments is allowed by the GHC because Haskell is a purely function programming language. As such the evaluation of an argument cannot have any side effects in Haskell and it is safe to reorder them arbitrarily. In our setting, on the other hand, the entire idea of the plugin is to allow for side effects. In consequence, a mechanism similar to the unboxing performed by the GHC cannot be adopted by the plugin without sacrificing the semantics of the program. Nevertheless, we argue that since the plugin is built around the idea of non-strict evaluation the user anticipates that the order of evaluation of arguments is not fixed. The degree to which the reordering of binds actually changes the semantics of the program depends largely on the specific monad that is used. In case of nondeterminism, for example, the reordering merely changes the order in which the results are produced. Thus, we can still safely integrate the transformation presented in this subsection into the Curry GHC Language Plugin. Whether the consequences of reordering operations are compatible with other instances of the plugin has to be carefully considered on an individual basis.

Before we conclude this subsection, let us demonstrate the transformation on a real example function again. This time we consider the function `fst` that projects a pair onto its first component. The lifted version of this function is shown on the left of the listing below. We suppose that the monad around the function has already been removed using the technique from the previous subsection. On the right side, we see the worker and wrapper functions that are obtained by applying the transformation presented in this subsection.

```
fstND :: Nondet (a, b)ND -> Nondet a
fstND = \mp -> mp >>= \p ->
  case p of { (x, _)ND -> x }
```

```
fstND :: Nondet (a, b)ND -> Nondet a
fstND = wrapStrict fstND_worker
      = \mp -> mp >>= \p -> fstND_worker p

fstND_worker :: (a, b)ND -> Nondet a
fstND_worker
  = unwrapStrict (\mp -> mp >>= \p ->
        case p of { (x, _)ND -> x })
  = \q -> return q >>= \p ->
        case p of { (x, _)ND -> x }
```

As we can see, the code still contains a `return` of the strict argument and a bind is needed in the worker to take the actual value out of the monad. Therefore, it seems as if we gained nothing by applying the transformation. On the contrary, there is now one bind more than before because the wrapper also binds the argument. However, the bind and `return` in the worker can be removed due to the left identity law of the monad. We do not have to apply the monad law ourselves, though, since the GHC is able to apply term rewriting rules (Peyton Jones, Tolmach, and Hoare, 2001). The plugin contains a `RULE` pragma conceptually identical to the one depicted below in order to instruct the GHC to simplify such cases.

```
{-# RULES "bind/return" forall f x. return x >>= f = f x #-}
```

After applying the term rewriting rule to the worker function $\texttt{fst}^{\texttt{ND}}_{\texttt{worker}}$, we obtain the following worker function that is exactly equivalent to the original unlifted function `fst` except for the lifted tuple and return type.

```
fstND_worker :: (a, b)ND -> Nondet a
fstND_worker = \q -> case q of { (x, _)ND -> x }
```

3.3.3 Sharing Strict Arguments

Until this point, we have not yet addressed the most severe cause of performance degradation that we identified in Section 3.1. In this subsection, we will present an extension to our previous transformation that can avoid the performance overhead of the `share` operator in cases that do not require deep sharing.

Consider a function that is strict in its argument and uses that argument multiple times. The lifted version of the function will have the following shape after we remove the outermost monad.

```
f :: Nondet A -> Nondet B
f mx = share mx >>= \sx -> sx >>= \x -> f' x x
```

If we now optimize the function as shown in the previous subsection, we obtain a worker function as shown below. Crucially, the rewrite rule that we relied upon to remove the bind and `return` from the worker function does not apply anymore because the `share` gets in the way.

```
f_worker :: A -> Nondet B
f_worker y = share (return y) >>= \sx -> sx >>= \x -> f' x x
```

At first glance, sharing the computation seems to be redundant since the argument has already been evaluated. However, a value of type A can still contain thunks that are unevaluated and therefore need to be shared. Nevertheless, the first step to our solution involves dropping the `share` of strict arguments from the worker altogether which yields the following (incorrect) code.

```
f_worker :: A -> Nondet B
f_worker y = return y >>= \x -> f' x x
```

As explained already, this solution is incorrect because nested values still need to be shared, i.e., we need to traverse the data structure and invoke `share` on every contained value. The `shareArgs` function from the **Shareable** type class has exactly these semantics. The result is the following final definition of the worker function.

```
f_worker :: A -> Nondet B
f_worker y = shareArgs y >>= \y' -> return y' >>= \x -> f' x x
```

To understand why this definition is superior to calling `share` directly, consider the following **Shareable** instances of a primitive and a composite data type.

```
instance Shareable Nondet Int where
  shareArgs = return

instance (Shareable Nondet a, Shareable Nondet b) => Shareable Nondet (a, b)^ND where
  shareArgs (a, b)^ND = (,)^ND <$> share a <*> share b
```

The **Shareable** instance of primitive types like **Int** defines `shareArgs` just as a synonym of `return`. In this case the GHC can inline the definition of `shareArgs` and apply the left identity law to effectively remove the sharing. For other data types such as lists or tuples, the GHC cannot perform such an optimization. The GHC might still inline `shareArgs` but the resulting code will contain recursive calls to `share`.

To realize this extension for the removal of `share`, we fortunately do not have to change our existing transformation. Instead, we can simply add another rewrite rule that replaces a `share` of a `return`ed value by a `shareArgs` followed by a `return` of the shared argument. The rewrite rule used by the plugin is conceptually equivalent to the one depicted below.

```
{-# RULES "share/return" forall f x. share (return x) >>= f = shareArgs x >>= f . return #-}
```

3.3.4 Unlifting Results

All of our current transformations have in common that the return type of the worker functions is still lifted into the monad. This is a direct result of the fact that we strive to use the GHC as much as possible to simplify the code for us. Namely, the right-hand sides of our worker functions never refer to any other worker function directly but only the corresponding wrapper functions. The return types of wrapper functions must be lifted in order to be compatible with the plugin's invariants. Therefore, it is very difficult to get rid of the monad without performing inlining ourselves. For the approach that we are going to present in this subsection, we again rely on the GHC's rewriting mechanism.

Suppose there exists an operation `unsafeBind :: Nondet a -> a` that extracts the value returned by a computation or throws an error at runtime in case the computation is effectful.[3] Then we can specify the following rule that eliminates a `return`.

```
{-# RULES "unsafeBind/return" forall x. unsafeBind (return x) = x #-}
```

To insert an `unsafeBind`, we formally define the following mappings between the types A `-> Nondet` B and A `->` B.

```
unwrap_Result :: (A -> Nondet B) -> A -> B
unwrap_Result f x = unsafeBind (f x)

wrap_Result :: (A -> B) -> A -> Nondet B
wrap_Result f x = return (f x)
```

Under the condition that the evaluation of the transformed computation is devoid of ambient effects, we expect the definitions of $wrap_{\texttt{Result}}$ and $unwrap_{\texttt{Result}}$ to satisfy the second worker/wrapper assumption (WW2) without proof. Unfortunately, it is very difficult if not undecidable to tell at compile time whether this condition is satisfied or not. Instead, we consider a weaker condition that is easier to check but only approximates the notion of effect-freeness. For the purposes of this thesis, we say that a function is effectful if it satisfies one of the following criteria and effect-free otherwise.

1. Intrinsically effectful functions that are built into the plugin such as `(?)` are effectful.
2. A function whose binding's right-hand side contains an occurrence of an effectful function is in turn also effectful.

While an analysis that checks this condition is easy to implement (see Section 4.2) and has been used by other Curry implementations successfully (Hanus and Skrlac, 2014), the guarantees provided by such an analysis are too weak to justify the insertion of `unsafeBind` in our setting. Recall for example the final definition of the worker function for `fst` presented in Section 3.3.2. Based on our definition, the function is effect-free. Thus, one might suppose that it is safe to remove the `Nondet` from the return type by inserting `unsafeBind` as shown below.

[3] See Appendix D for the implementation of `unsafeBind` used by the Curry GHC Language Plugin.

```
fst^ND_worker :: (a, b)^ND -> a
fst^ND_worker q = unsafeBind (case q of { (x, _)^ND -> x })
```

However, this is not the case since `fst` might be applied to a pair whose first component is the result of an effectful computation, e.g., `fst (0 ? 1, 2)`. Which additional criteria have to be met in order to safely perform this optimization is a question that we leave open for future research.

3.3.5 Final Transformation Scheme

Until now we have only seen one piece of the worker/wrapper transformation at a time. Additionally, we have considered unary functions only. To wrap up the explanation of the optimization approach, we summarize the final transformation scheme for functions of an arbitrary arity in this subsection.[4]

Given an n-ary function declaration before lifting as shown below on the left, the plugin usually produces a function of the form shown on the right. Furthermore, we suppose that the function is strict in m arguments with indices $S = \{ s_1, \ldots, s_m \} \subseteq \{ 1, \ldots, n \}$. For all $i \in \{ 1, \ldots, n \}$ let y_i denote a fresh variable if $i \in S$ and define $y_i := x_i$ otherwise.

```
f :: X_1 -> X_n -> R
f x_1 ... x_n = rhs
```

```
f^ND :: Nondet (X_1^ND --> ... --> X_n^ND --> R^ND)
f^ND = return $ Func $ \x_1 ->
         ⋱
           return $ Func $ \x_n -> rhs^ND
```

When the worker/wrapper transformation is used, we want the plugin to generate a wrapper function similar to the lifted function above, but instead of the right-hand side rhs^{ND} we want to have a call to the worker function f^{ND}_{worker}. Additionally, the wrapper binds the arguments x_i for $i \in S$ to fresh variables y_i before the application of the worker function. In total, the wrapper should look as follows.

```
{-# INLINE f^ND #-}
f^ND :: Nondet (X_1^ND --> ... --> X_n^ND --> R^ND)
f^ND = return $ Func $ \x_1 ->
         ⋱
           return $ Func $ \x_n ->
             x_{s_1} >>= \y_{s_1} ->
               ⋱
                 x_{s_m} >>= \y_{s_m} -> f^ND_worker y_1 y_2 ... y_n
```

Since this kind of function might exceed the maximum size the GHC is usually willing to inline, we explicitly direct the GHC to always inline the wrapper by adding an **`INLINE`** pragma.

[4] In Appendix D the proposed extension of the transformation scheme to effect-free functions is outlined.

Finally, the worker function is an n-ary function whose i-th argument's type is wrapped with `Nondet` if and only if $i \notin S$. For this purpose, we define for all $i \in \{1, \ldots, n\}$ the following meta type variable.

$$Y_i^{\mathsf{ND}} := \begin{cases} X_i^{\mathsf{ND}} & \text{if } i \in S \\ \texttt{Nondet}\ X_i^{\mathsf{ND}} & \text{if } i \notin S \end{cases}$$

Since the right-hand side rhs^{ND} still refers only to lifted arguments, we introduce local variable binders that wrap the strict arguments with `return`. We rely on the GHC to inline these local variables and apply rewrite rules to eliminate the `return`s where possible.

```
f^ND_worker :: Y^ND_1 -> ... -> Y^ND_n -> Nondet R^ND
f^ND_worker y_1 ... y_n =
  let x_{s_1} = return y_{s_1}
      ⋮
      x_{s_m} = return y_{s_m}
   in rhs^ND
```

In the next chapter, we will explain how we implemented this transformation in the plugin.

Chapter 4

Implementation

In the preceding chapter, we layed out a strategy for how we can employ the worker/wrapper transformation to optimize monadically lifted code. In this chapter, we shift our focus towards the practical implementation of this optimization within the plugin. In the first section, we describe the implementation of an analysis that collects strictness information about functions in the translated program. This information is needed to steer the optimization process. Additionally implemented analyses are presented in the two subsequent sections. The effect analysis is part of the experimental extension to the optimization discussed in Section 3.3.4. The improved sharing analysis aids in the avoidance of excessive sharing. In the last two sections, we describe the implementation of the actual worker/wrapper transformation. Unlike in our theoretical considerations, the optimization does not directly operate on the monadically lifted code. We first describe how unlifted function bindings can be split into a worker and a wrapper function. Finally, we outline modifications to the monadic lifting that respects the distinction between worker and wrapper functions. The general architecture of the plugin after our implementation is depicted in Figure 4.1. Except for the constraint solver plugin, all parts of the pipeline after type checking are involved in the optimization process.

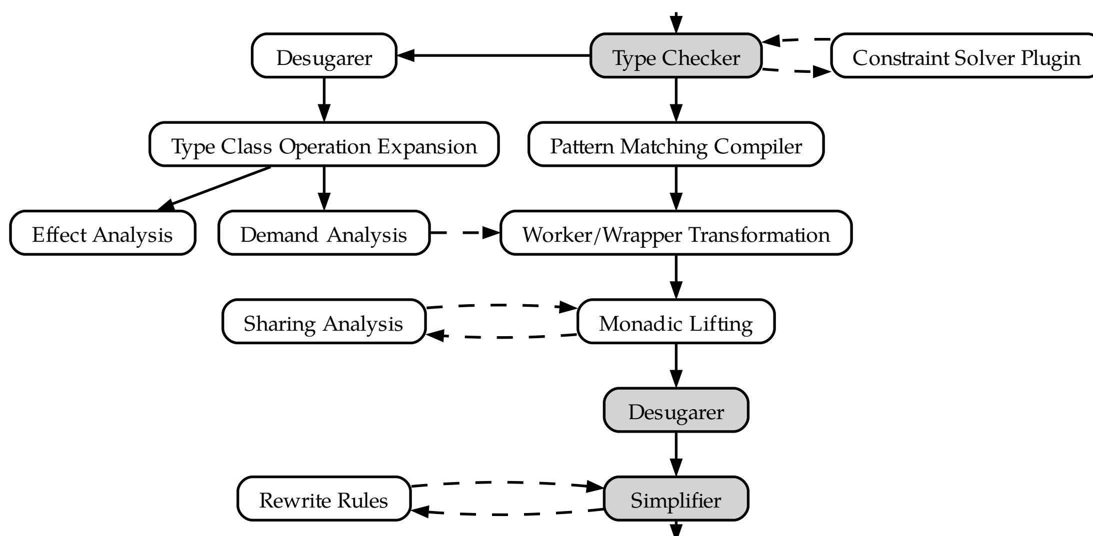


Figure 4.1. Overview of the updated plugin architecture. Gray nodes are part of the regular GHC compilation pipeline. White nodes are executed as part of the plugin's translation. Dashed lines represent the flow of information while solid lines trace processing of the AST. Only phases after type checking are shown.

4.1 Demand Analysis

A key component of our optimization is the monadic unwrapping of strict arguments. Therefore, we need to know which arguments of a function are strictly evaluated. A naive approach could be to look for pattern matches of the argument on the right-hand side. For instance, we know that the argument of the following function is strictly evaluated because it is scrutinized by the `case` expression.

```
null :: [a] -> Bool
null xs = case xs of
  [] -> True
  _  -> False
```

However, this approach falls short if the scrutinee is a more complex term such as `id xs`. Furthermore, the entire `case` expression could be in a position that is never even forced. Concluding that the argument would still be strict is erroneous in such a situation. Therefore, we need a more sophisticated mechanism to determine strictness. As we mentioned in Section 3.2.2 already, the GHC optimizes functions with strict arguments itself. In consequence, the GHC also has to perform an analysis to determine which arguments are strict. Since the GHC's analysis is even more capable, it is called a *demand analysis*. In the following, we are first going to explain how the GHC keeps track of the demands that are placed by a function on its argument. Next, we outline how we managed to reuse the GHC's demand analysis for our purposes. We will notice, that when we apply the analysis on its own is not sufficient in all cases. The remainder of the section is therefore concerned with preprocessing steps that are necessary to cover all edge cases.

The GHC's demand analysis is described in an unpublished draft paper by Sergey, Peyton Jones, and Vytiniotis (2017). Additional information about the GHC's demand analysis can be found in the GHC user guide entry for the `-fstrictness` flag[1].

4.1.1 Demand Signatures

The GHC's demand analysis records the strictness information in a so-called *demand signature*. Demand signatures are represented by the `StrictSig` type. A demand signature mainly consists of a list of demands that are placed on the function's arguments. Additionally, demand signatures contain information about the divergence of the function as well as the demands that are placed on free variables in the environment. The divergence information is of no relevance to us and can be ignored. Furthermore, we only consider demand signatures that are *closed*, i.e., do not have environmental demands. Syntactically, the closed demand signature of a non-diverging n-ary function is denoted as follows where d_1 through d_n denote the argument demands.

$$<d_1><d_2><d_3>\ldots<d_n>$$

The list of demands contains exactly one entry per function argument. The type of the function is

[1] `https://downloads.haskell.org/~ghc/9.2.1/docs/html/users_guide/using-optimisation.html#ghc-flag--fstrictness`

irrelevant in this regard. An argument counts as an argument if there is a lambda abstraction that binds it.

A demand consists of two parts: a cardinality interval c and a sub-demand s and is denoted $d = cs$. The cardinality interval provides a lower and an upper bound for the number of times the argument is evaluated at runtime. For the cardinality, the compiler only distinguishes whether the argument is evaluated never (0), exactly once (1) or an unbounded number of times (ω). These cardinalities can be combined to form the intervals shown in Table 4.1. The cardinality intervals `1` and `S` are the only ones we are interested in since they indicate that an argument is evaluated strictly. We will also see the `L` interval commonly because it is used as a default value when the analysis cannot provide any guarantees. The interval `M` does not carry much meaning for us because there is still the possibility that the argument is not evaluated at all. Even though it would be easy to remove unused arguments with cardinality interval `A` from worker functions, we do not have to implement this optimization ourselves because the GHC's own worker/wrapper transformation will take care of that. Therefore, the `A` cardinality interval is also irrelevant to us.

Table 4.1. All possible non-empty cardinality intervals that can be formed. The second column contains the set of cardinalities that are contained in the interval. The syntax in the third column is used to denote the interval in demand signatures. The last column contains a description of the interval.

Interval	Set of Cardinalities	Notation	Description
$[0, 0]$	$\{0\}$	`A`	**Absent**: The argument is never evaluated
$[0, 1]$	$\{0, 1\}$	`M`	**Maybe**: The argument is conditionally evaluated
$[1, 1]$	$\{1\}$	`1`	**Once**: The argument is evaluated exactly once
$[1, \omega]$	$\{1, \omega\}$	`S`	**Strict**: The argument is evaluated at least once
$[0, \omega]$	$\{0, 1, \omega\}$	`L`	**Lazy**: The argument's evaluation is unbounded

The sub-demand specifies the depth of the evaluation. Sub-demands are of very little interest to us because our transformation does not split product types unlike we have seen in the example of the GHC's worker/wrapper transformation in Section 3.2.2. Nevertheless, we are going to briefly explain the different kinds of sub-demands below before we move on to the actual implementation of the demand analysis.

- ▷ **Product Sub-Demands** specify the demands that are placed on the fields of a value whose type has a single constructor, i.e., that is isomorphic to some tuple type. A product sub-demand has the shape `P(`d_1`, ..., `d_n`)` where d_i is the demands placed on the i-th field. There is no equivalent for sum types.

- ▷ **Polymorphic Sub-Demands** specify the cardinality interval of all contained values. The polymorphic sub-demand c is equivalent to an infinitely nested product sub-demand `P(`c`P(`c`P(...)), ...)`.

- ▷ **Call Sub-Demands** specify how often a function is called and how the result is demanded. A call sub-demand has the shape `C`c`(`d`)` where the cardinality interval c specifies how often the function is called and the sub-demand d specifies how the result is demanded.

4.1.2 Using the GHC Demand Analysis

The major obstacle to using the GHC's demand analysis is that the GHC performs the analysis on its intermediate representation GHC Core while the plugin operates on the Haskell AST. Therefore, we need to translate the Haskell AST into GHC Core first. We can use the GHC's own desugarer for this purpose. The plugin actually desugars the program prior to lifting already. The result of this operation is unused so far. The reason why the plugin invoked the desugarer was to trigger the GHC's checks for compilation errors. In principle, we can just use the result of desugaring for our purposes. However, the GHC's desugarer applies some trivial simplifications to the program. For example, local bindings that are used only once are directly inlined. This is problematic for our purposes because we want to apply our worker/wrapper transformation to such local functions as well and therefore need their strictness information. In fact, it is very common in Haskell to define the actual implementation of a function locally and have the top-level function be a wrapper that simply calls the local function. This approach is used for functions with accumulator argument for instance. To prevent the desugarer from performing any premature optimizations among these lines, the plugin pretends that a `NOINLINE` pragma has been set for every identifier when invoking the desugarer.

Now that we desugared the program to a `CoreProgram`, we can apply the GHC's demand analysis on it. To do so we invoke the following function defined in `GHC.Core.Opt.DmdAnal`.

```
dmdAnalProgram :: DmdAnalOpts -> FamInstEnvs -> [CoreRule] -> CoreProgram -> CoreProgram
```

The first argument contains additional options for the analysis. In version 9.2.1 of the GHC, the only field of `DmdAnalOpts` is the flag `dmd_strict_dicts`. This option controls whether the demand analysis considers type class dictionary arguments to be strict or not. We do not care about the strictness of such arguments. Thus, we have arbitrarily chosen to set the flag to `False`. The `FamInstEnvs` in the second argument contains information about `type` and `data family` instance declarations and can be obtained via `tcGetFamInstEnvs` from the current environment. Finally, the list of `CoreRule` represents `RULES` pragmas that are in scope and is left empty, because we do not want the analysis to take rewrite rules into account.

The result of demand analysis is a `CoreProgram` where the GHC has attached demand signatures to identifiers. We continue by extracting these demand signatures from the `CoreProgram` and storing them in a `DemandMap`. This data type is an abstraction that we build on top of an efficient map data type provided by GHC. To populate the map, we first enumerate all binders using the `syb` library and associate their identifier with the demand signatures. We have to be careful to use the correct identifiers. After type checking all functions have two kinds of identifiers in the Haskell AST: a monomorphic and a polymorphic identifier. This is even the case when the function is not polymorphic at all. During our transformation, we need to have access to the demand signature regardless of which identifier we are using. After desugaring, the distinction between monomorphic and polymorphic identifiers only remains in case of mutually recursive bindings. Therefore, we artificially extend the demand map to include entries for the missing identifiers.

In principle, we are done with the implementation of the demand analysis itself at this point. However, the analysis does not yet produce the desired results. For the remainder of the section, we are going to tackle the remaining issues by defining preprocessing steps that need to be applied between the desugaring and demand analysis.

4.1.3 Analysis of Function Arities

The first problem we encounter when running the demand analysis is that the demand signatures are still empty afterward. This peculiar behavior is a consequence of the fact that we invoke the demand analysis right after the conversion to Core. In the actual compilation pipeline, the demand analysis is interleaved with other compilation passes of the GHC. Therefore, the analysis makes assumptions about the program that are not necessarily fulfilled in our case. One of these assumptions is that the identifiers of function bindings are annotated with their arity. The demand analysis will only consider the demands of functions up to that point. Initially, the desugarer sets the arity of all functions to zero. The GHC then analyzes the right-hand side to determine the actual number of arguments a function can be applied to without doing any work. We will have to replicate this step to satisfy the demand analysis. Again, we try to reuse the procedure that the GHC performs on its own. Unlike the demand analysis, there is no clear entry point for the arity analysis, though. The module `GHC.Core.Opt.Arity` contains a variety of utility functions regarding function arities and eta-expansion. The most promising function is `manifestArity :: CoreExpr -> Arity`. This function counts the number of value lambda abstractions a Core expression starts with. Conveniently, the function can look through casts and ticks which are sometimes inserted by the desugarer. Additionally, the Core Code can sometimes contain `let` expressions for dictionary binders before the lambda abstractions and we need to look through these as well. Unfortunately, `manifestArity` cannot do so. Thus, we had to copy the source code and add the missing rule for `let` expressions manually. While this is not ideal, the function is not very complex either.

A detail that we had to take care of during arity analysis again regards the distinction between monomorphic and polymorphic identifiers. The monomorphic identifier is used for the binding that contains the real implementation of the function. Therefore, its arity is determined correctly. The binding of the polymorphic identifier on the other hand is usually just a wrapper that binds some type and dictionary arguments and then returns the monomorphic function. The real arguments are therefore not counted towards the arity of the polymorphic identifier. We solve this problem by adding the monomorphic identifier's arity to the polymorphic identifier's arity in a postprocessing step.

Additional problems can arise when the arity of a function in the desugared program differs from the arity of the function in the original Haskell program. Usually, this does not happen because we do not invoke the GHC's simplifier and the desugarer also performs no eta-expansion or reduction by itself. The only exception are tuple sections[2] which are desugared to lambda abstractions. The following binding therefore causes an arity mismatch.

```
withZero :: a -> (Int, a)
withZero = (0,)
```

We solved this issue by extending the preprocessing of the plugin that usually compiles pattern matching. We now also rewrite tuple sections in terms of lambda abstractions. The example above is therefore rewritten to the following.

```
withZero :: a -> (Int, a)
```

[2] Enabled by the `TupleSections` language extension.

```
withZero = \x -> (0, x)
```

Even though this preprocessing step fixes the aforementioned problem, the example of tuple sections demonstrates the fragility of our approach.

4.1.4 Analysis of Imported Functions

After fixing the arity analysis, the demand analysis finally produces some results. Unfortunately, we encounter problems as soon as imported functions are involved. Consider, for example, the following definition that uses the function `null` from the `Prelude`.

```
safeHead :: [a] -> Maybe a
safeHead xs = if null xs then Nothing else Just (head xs)
```

Since `null` is strict in its argument and occurs in a position where its evaluation is demanded, we expect the demand analysis to conclude that `safeHead` is strict in its first argument as well. Even though, we can verify that the demand analysis correctly computed a demand signature of `<1L>` for `null` during the compilation of the `Prelude`, the demand signature of `safeHead` is still `<L>`. The reason is that the lifted version of `null` has the following shape.

```
null^ND :: Nondet ([a]^ND --> Bool)
null^ND = return $ Func $ \xs -> ...
```

Evidently, this binding is not strict in its first argument anymore as there is no argument left to be strict in. The actual function has been hidden behind the monad and `Func` constructor. The importing module "sees" only this lifted definition of `null` and not the original version. This problem is related to the reason why the plugin needs to intervene during type checking. Unlike the original type, the original demand signature is not restorable from the lifted version, though. In consequence, we need a mechanism to memorize the original demand signature and look up this information before demand analysis.

To store the original demand signature we add an annotation to the function in the form of an `ANN` pragma. Such pragmas can be used to attach arbitrary Haskell values to top-level identifiers provided the type implements the `Data` type class, i.e., is serializable. Unfortunately, the `StrictSig` type does not have a `Data` instance as the GHC uses a custom serialization mechanism internally. We can also not define such an instance ourselves[3], because not all constructors of the contained `SubDemand`s are exported by the GHC. Even if it had such an instance, using the `StrictSig` data type inside of the annotation would make it difficult to manually annotate a function. Such manual annotations are necessary to correctly specify the demands of built-in functions. Therefore, we decided to use the following data type for annotations where the `String` field contains the pretty-printed demand signature. The format is the same as presented in Section 4.1.1. The `Data` type class can be derived automatically using the `DeriveDataTypeable` language extension.

[3] For example by using the `StandaloneDeriving` and `DeriveDataTypeable` language extensions.

```
newtype DemandSignatureAnnotation = OriginalDemandSignature String
  deriving Data
```

While producing the string representation of a demand signature is trivial by using the GHC's `Outputable` type class, the question that remains is how we can reverse this process. The GHC does not have this functionality built-in because a binary format is used for its own serialization mechanism whereas the `Outputable` type class produces output that is intended for humans to read. Therefore, we had to implement our own parser for demand signatures.

The parser is implemented using the `parsec` parser combinator library. The only problem that arose during the parser's implementation is again that the constructors of `SubDemand` are not accessible. Because our transformation does not use sub-demands anyway, we decided to simply ignore such sub-demands during parsing, inserting the polymorphic sub-demand `L` in their place. While a sub-demand of `L` is never wrong, we have to keep in mind that as a result, demand analysis might sometimes find weaker demands than expected.

Given the annotation and parser, we can add a preprocessing step, where we look up the annotation of each imported identifier in the program and parse the contained string to restore the original demand signature of the identifier. Additionally, we need a postprocessing step that adds suitable annotations based on the calculated demands. For example, the following pragma is added to the lifted version of `null`.

```
{-# ANN null^ND (OriginalDemandSignature "<1L>") #-}
```

4.1.5 Analysis of Type Class Operations

So far our presentation focussed on the analysis of regular function bindings. In this section, we direct our attention to the analysis of type class operations. In Section 2.2.3 we learned about dictionary passing style and how type class instances are represented in GHC Core. Let us recapitulate how the `Num` type class and a corresponding instance for Peano numbers might be represented in GHC Core. The original definition is shown below on the left. On the right side, we see show the corresponding GHC Core code produced by the desugarer.

```
class Num a where
  (+) :: a -> a -> a
  ...
```

```
data Num a = Num (a -> a -> a) ...

(+) :: Num a -> a -> a -> a
(+) (Num add) = add
```

```
instance Num Nat where
  O + n = n
  S m + n = S (m + n)
  ...
```

```
d_Num,Nat :: Num Nat
d_Num,Nat = Num (+_Nat) ...

(+_Nat) :: Nat -> Nat -> Nat
O +_Nat n = n
(S m) +_Nat n = S (m +_Nat n)
```

The type class itself is represented as a product data type with one field per operation. The operations are selectors for the corresponding field. Their type class constraint has become an additional argument for the instance's dictionary. For instance, a top-level binding $d_{Num,Nat}$ has been created that constructs the dictionary. The implementation of the operation was also lifted to top-level and is referred to in the corresponding field of the dictionary constructor.

Since the demand analysis operates on GHC Core, the type class operation $(+_{Nat})$ is treated as a regular function binding such that the analysis works as expected, i.e. the demand analysis correctly concludes that the first argument of $(+_{Nat})$ is strictly evaluated. While the analysis of the declaration of type class operations is not problematic, the demand analysis struggles with the analysis of functions that use type class operations. Compare, for instance, the following example function and its desugared version.

```
inc :: Nat -> Nat
inc n = n + (S O)
```

```
inc :: Nat -> Nat
inc n = (+) @Nat d_Num,Nat n (S O)
```

While the function is guaranteed to evaluate its argument strictly, demand analysis assigns the demand signature `<L>` to `inc`. The reason is that the right-hand side does not mention the implementation $(+_{Nat})$ of the operation for the **Nat** data type but the dictionary field selector `(+)`. The selector has demand signature `<1P(1C1(1C(1L)), ...)>`. The cardinality indicates that the first argument (the dictionary) is strictly evaluated. The sub-demand specifies that the first field of the dictionary is also strictly evaluated, applied once to exactly two arguments and the return value is used in a strictly demanded position. Most importantly, there is no information about how the non-dictionary arguments are demanded. The reason is that the operation can be arbitrarily implemented for different data types.

The GHC usually does not encounter the problem for the same reason that the analysis can safely assume that the arities of all functions have been determined already. Since the demand analysis is interleaved with the simplifier and the simplifier inlines the value of the field selected by `(+)`, the GHC can "see" the actual implementation. In our case, the AST is not processed any further after desugaring. At first, we tried to reuse GHC's simplifier to inline the type class operations for us. Unfortunately, we did not find a way to restrict the simplifier such that it performs no other optimizations. In consequence, we had to write our own logic to extract the demand signature from the implementation. This process is quite complicated. Furthermore, we have to distinguish whether the instance is declared in the same module, another lifted module or is built-in.

Let us first consider the case that `inc` and the `Num` instance for `Nat` are declared in the same module. In a preprocessing step, we search for all functions application expressions op_i @τ_1 ... @τ_k d_1 ... d_m x_1 ... x_n where op_i is a dictionary field selector for the i-th operation of some k-parameter type class. We then look up the `Unfolding` of d_1 and check whether it is a dictionary function, i.e., a `DFunUnfolding`. If it is, we syntactically replace op_i @τ_1 ... @τ_k d_1 with the dictionary constructor's i-th argument. Most often, the evidence variable d_1 does not directly unfold to the dictionary constructor, though. As a result of a process known as *zonking*, the GHC usually introduces intermediate evidence binders as shown below.

```
inc :: Nat -> Nat
inc n =
  let ev = d_{Num,Nat}
   in (+) @Nat ev n (S O)
```

For this reason, we have to recursively follow such a chain of `CoreUnfolding`s until we find a `DFunUnfolding`. In our example, the eventual result of this whole procedure is that `inc` directly applies the $(+_{Nat})$ function. As a result, we can continue with the usual demand analysis.

Unfortunately, the process described above does not work if the instance is imported from another module. Unlike the unfolding of local instances that have not passed through the plugin yet, the unfolding of imported type class instances contains lifted code. Therefore, we cannot simply expand the type class operation in such cases. However, there is also no need for the demand analysis to know the exact implementation of the type class operation. Similar to regular imported functions, the analysis only needs to know the demand signature of the operation. Thus, we first look up the demand signature by following the chain of unfoldings and parsing the `OriginalDemandSignature`. Then we attach this demand signature to the selector function's identifier. Additionally, we have to remove the information that the identifier refers to a dictionary field selector since the demand analysis treats such functions differently.

While this approach works in theory, in practice we are confronted with the problem that the plugin removes the unfolding of dictionary functions as part of `liftInstance` in `Plugin.LanguagePlugin.Lift.ClsInst`. For other kinds of functions, this is not a problem, because the GHC recalculates the unfolding in a later step of the compilation pipeline. In case of dictionary functions, however, the type checker is responsible for adding the `DFunUnfolding`. Since the type checker is not executed again, we have to recreate the unfolding ourselves. However, it turns out that the dictionary functions are not compatible with this representation anymore after lifting. The reason is that the plugin inserts evidence binders for `Shareable` constraints. Therefore, we need an indirection via a `CoreUnfolding`, i.e., need to add a wrapper function that applies the dictionary function. Since implementing this requires almost a complete rewrite of the dictionary lifting logic and custom type class instances do not properly work in the current version of the plugin anyway, we decided to leave this task as future work.

Nevertheless, we need to be able to analyze operations from imported type class instances at least for built-in types as those are among the most frequently used operations. Since built-in instances are not passed through the plugin, their unfoldings are not erased. Thus, we only need to annotate the built-in operations with their `OriginalDemandSignature`. At first we tried to directly attach the annotation to the declaration of the operation in the instance as shown below for the built-in instance

of the lifted **Num** class for **Int**. Note that the demand signature specifies demands for three arguments because there is an additional argument for the dictionary in the corresponding Core code.

```
instance Num^ND Int where
  {-# ANN (+^ND) (OriginalDemandSignature "<L><1L><1L>") #-}
  (+^ND) = liftNondet2 (+)
```

However, this causes a syntax error because the GHC only allows **ANN** pragmas on top-level declarations. Since we cannot directly annotate the underlying top-level binding generated by the desugarer, we have to define our own top-level binding to carry the annotation. We then implement the operation in the instance as an alias for this new top-level function.

```
instance Num^ND Int where
  {-# INLINE (+^ND) #-}
  (+^ND) = add^ND_Int

{-# INLINE [0] add^ND_Int #-}
{-# ANN add^ND_Int (OriginalDemandSignature "<L><1L><1L>") #-}
add^ND_Int :: Nondet (Int --> Int --> Int)
add^ND_Int = liftNondet2 (+)
```

Additionally, **INLINE** pragmas (which are allowed in instance declarations contrary to **ANN** pragmas) are added to both functions. The **INLINE** pragma on the $\texttt{add}^{ND}_{\texttt{Int}}$ function ensures that the indirection does not introduce an additional function call at runtime. The **INLINE** pragma on the $(+^{ND})$ operation, on the other hand, prevents the GHC from inlining the definition of $\texttt{add}^{ND}_{\texttt{Int}}$ prematurely. Otherwise, the unfolding of $(+^{ND})$ would contain `liftNondet2 (+)` instead of $\texttt{add}^{ND}_{\texttt{Int}}$ but we need to see that identifier in order to extract the **OriginalDemandSignature** annotation.

If we consequently apply this pattern to all built-in type class operations, we can analyze them correctly. We believe that once the unfolding of lifted dictionary functions is fixed, we can analyze operations from arbitrary imported type class instances with this method. Nonetheless, the whole process of unfolding the dictionaries is extremely sensitive to the internal workings of the GHC and is likely to break with a future update.

This concludes our description of the demand analysis implementation. In the last subsection, we are going to point out one final major limitation of our approach that cannot be addressed without incorporating the GHC's simplifier into the analysis pipeline.

4.1.6 Analysis of Higher-Order Functions

As Haskell and Curry are functional programming languages, a central aspect of them is the concept of higher-order functions. There are currently two problems with demand analysis when higher-order functions are involved. This first problem was implicitly discussed in Section 4.1.4 already and is not of fundamental nature. Our parser for demand signatures currently ignores sub-demands since

there is no straightforward way to create instances of the underlying data type. Usually, a call sub-demand tells the GHC when the higher-order function calls the given function strictly. For example, the following function's demand signature of <1C1(L)><L> expresses that the argument f is strictly evaluated and the resulting function applied once whereas we cannot tell anything about the demand placed on x because we do not know the demand signature of f itself.

```
apply :: (a -> b) -> a -> b
apply f x = f x
```

Now consider the following artificial example function that invokes apply.

```
increase_if :: Bool -> Int -> Int
increase_if flag n = apply (\n' -> if flag then n' + 1 else n') n
```

If apply and increase_if are defined in the same module, the demand analysis can take the call sub-demand of apply into consideration and thus knows that flag is strictly evaluated. When apply is imported, the parser discards the call sub-demand and therefore demand analysis cannot tell that increase_if is strict in flag. While this problem could be easily solved by using a more sophisticated mechanism for restoring demand signatures of imported functions, higher-order functions pose another problem that is more fundamental to our approach.

In the preceding sections, we were already confronted multiple times with challenges that arise since we apply the demand analysis out of the context it was originally designed for. Consider again the example of increase_if. Since this function is strictly evaluating the lambda abstraction which in turn is strict in n[4], we might conclude that the entire function is strict in its second argument. However, this is not the case because demand analysis never looks into the definition of a function and therefore does not know that n is eventually passed as the argument n' to the lambda abstraction. To analyze these kinds of situations, the demand analysis usually relies on the simplifier to inline the definition of the higher-order function.

An example, where this problem can be observed in practice are sections, i.e., partially applied infix operators. While we discovered in Section 4.1.3 that tuple sections are desugared to lambda abstractions, sections of infix operators are desugared to applications of built-in functions leftSection and rightSection, respectively. The leftSection function corresponds exactly to the apply function while rightSection is of type (a -> b -> c) -> b -> a -> c and basically just applies the given function with its arguments in reverse order. Since sections are defined in terms of higher-order functions, the strictness information of the sectioned operator is lost.

While we could solve the problem of sections by desugaring them to lambda abstractions in the same way we did for tuple sections, this would not solve the general problem that higher-order functions cannot be properly analyzed without the simplifier. Unfortunately, we learned from our attempts in Section 4.1.5 that integrating the simplifier into our plugin is most likely not feasible because the program is technically speaking incorrect prior to lifting. Therefore, we leave this problem open for future investigation.

[4] A function is always strict in its return value since after β-reduction the returned value ends up in the currently forced position.

4.2 Effect Analysis

Before we continue with the implementation of the worker/wrapper transformation, we first discuss the remaining analyses that we implemented over the course of this thesis. In this section, we cover an analysis that approximates whether a function has an effect or not. This analysis is currently not used by any other part of the plugin but could be used in the future to automatically decide whether the use of `unsafeBind` is safe.

As there is no concept of an effect in Haskell, there is also no analysis in the GHC that we could repurpose for the plugin's effect analysis. Instead, our implementation is based on the determinism analysis presented by Hanus and Skrlac (2014) as an example for their Curry Analysis Server System (CASS). In CASS a program analysis is a mapping of functions into an *abstract domain*. Such an abstract domain is a type where each element represents a set of functions. For example, the determinism analysis is performed on the following abstract domain where a value of **`Det`** represents all functions that are guaranteed to produce at most one value when called on ground values whereas **`NonDet`** captures operations that *might* produce multiple results under this condition.

```
data DetDom = Det | NonDet
```

The analysis itself is a fixpoint computation. This means that the computation maintains a map from function identifiers to values of the abstract domain and updates this map iteratively until there are no changes anymore. In case of the determinism analysis, all functions are initially considered deterministic. In each iteration, there is a check for every function in the program whether the right-hand side contains the `(?)` operator, any free variables or a call to a function that has the abstract value **`NonDet`**. If this condition is satisfied, the abstract value of the function is set to **`NonDet`**. Otherwise, the value remains at **`Det`**. Since the set of functions whose abstract value is **`NonDet`** can only grow but the total number of functions in the program is finite, this kind of algorithm is guaranteed to terminate eventually. In the worst case, only one function's abstract value is set to **`NonDet`** in every iteration. In this case, the algorithm terminates after n iterations where n is the number of functions in the program. Since all functions have to be checked per iteration, the algorithm has a time complexity of $\mathcal{O}(n^2)$ in total.

In order to adopt this algorithm, we first need to define our own abstract domain for the distinction between effectful and effect-free functions. We are using the following data type for this purpose.

```
data EffectInfo = HasNoEffect | MayHaveEffect | HasEffect
  deriving (Eq, Ord, Bounded, Show, Data)
```

The **`HasEffect`** constructor roughly corresponds to **`NonDet`** and represents all functions whose evaluation *might* have an effect whereas **`HasNoEffect`** is analogous to **`Det`** and indicates that a function always evaluates without causing an effect. The additional **`MayHaveEffect`** constructor has no corresponding value in the **`DetDom`** domain of the original paper. Semantically, this value can be interpreted equivalently to **`HasEffect`**. The only distinction is that we do not want to include functions in **`HasEffect`** if the potential effect stems from the usage of a type class operation whose implementation is not statically known. This distinction helps us during development to test whether the effect analysis

correctly handles type class operations. For the expansion of type class operations, we reuse the logic from Section 4.1.5. As could already be seen in Figure 4.1, the handling of type class operations has therefore been pulled out into a common preprocessing step.

Analogously to the demand signature annotations from the previous section, we rely on `ANN` pragmas to remember whether a function has an effect or not across module boundaries. For this purpose, we reuse our data type of the abstract domain as an annotation. This is why we derived a `Data` instance for `EffectInfo` above. The built-in `(?)` operation is annotated with `HasEffect`. All other built-in operations do not have an annotation. The absence of the annotation is interpreted as if the function were annotated with `HasNoEffect`. When the analysis is used in the future, we might have to consider adding a `HasEffect` annotation to `failed` as well depending on the exact information needed. We suspect that this might be the case because `unsafeBind` expects exactly one result but `failed` produces none. Since the built-in `pE` function that is used for pattern matching failures is implemented in terms of `failed`, this would also require adding a `HasEffect` annotation to `pE`. For our current prototype, we decided to stay as close as possible to the original determinism analysis, though.

Before the actual analysis is started, the plugin collects all identifiers in the current module using `syb`, looks up their `EffectInfo` annotation and writes this data into a map that is used as the initial value of the fixpoint algorithm. For type class operations that have not been expanded in the preprocessing step, a value of `MayHaveEffect` is inserted into the map. In each iteration of the algorithm, we collect all identifiers that are used on the right-hand side of a function definition again using `syb`. Then we look up the corresponding `EffectInfo` in the map. Finally, we write back the `maximum` of these values or default to the "smallest" possible value, i.e., `HasNoEffect`, if there are none. To this end, we derived an `Ord` and `Bounded` instance for `EffectInfo`. The ordering of the elements imposed by the derived `Ord` instance is as follows.

$$\texttt{HasNoEffect} > \texttt{MayHaveEffect} > \texttt{HasEffect}$$

In effect, the abstract value associated with every function can only increase in each iteration giving us the same termination guarantee and runtime complexity as the original determinism analysis had.

4.3 Sharing Analysis

In Section 3.3.3 we presented a rewrite rule that replaces the `share` operator inside worker functions with `shareArgs` which the GHC subsequently eliminates for primitive data types. However, this rewrite rule is not applicable unless the argument is strict. Since the `share` operator is associated with a severe performance overhead, we strive to minimize the number of `share` operators that are inserted in the first place. The plugin currently inserts a `share` operator for every argument, variable pattern or local variable binding that occurs more than once on the corresponding right-hand side (or `in` expression in case of local variable bindings). To count the number of times a variable `v` occurs in an expression `e`, the plugin enumerates all variables `v'` in `e` with the same `Unique` as `v`. As shown below, this algorithm can be implemented using the `syb` library very concisely.

```
countVarOcc :: (Data a) => Var -> a -> Int
countVarOcc v e = length (listify (\v' -> varUnique v' == varUnique v) e)
```

A shortcoming of this approach is that it cannot tell whether the value will actually be used twice during the evaluation of `e` or the two occurrences reside in mutually exclusive branches of the computation. Consider, for example, the following function.

```
append :: [a] -> [a]
append xs ys = case xs of
  []      -> ys
  (x:xs') -> x : append xs' ys
```

The argument `ys` occurs twice on the right-hand side of `append`: once in the branch for the empty list and once in the branch for the non-empty list. Thus, the plugin will insert a `share` operator for `ys` even though a concrete invocation of `append` will never involve more than one usage of the argument. Therefore, we extended the sharing analysis of the plugin to account for such cases. The idea is still to count the number of variable occurrences. However, instead of collecting all variable occurrences upfront, we traverse the AST until we encounter an **`if`**, **`case`** or variable expression. For a variable expression, we simply compare the **`Unique`**s. For **`if`** and **`case`** expressions, we recursively count the variable occurrences in each branch of the expression and take the maximum of the results. For all other kinds of nodes, we sum up the counts from the child expressions.

Due to the need to traverse the AST, the implementation of this analysis is more involved than the previous one. By using the `syb` library, we can still implement the analysis in a generic way and do not have to deal with the complex structure of the AST too much. Particularly, we are using the `everythingBut` function to descend into the AST until we find a node of interest but no deeper. We cannot use `everything` because we want to use explicit recursion to count in the branches of **`if`** and **`case`** expressions. One challenge that we encountered during the implementation is that **`HsVar`**, **`HsIf`** and **`HsMultiIf`**[5] are part of the **`HsExpr`** type whereas the branches of all kinds of **`case`** expression[6] can be found in the **`MatchGroup`** type. We deal with this situation by combining multiple generic queries for the different types using the `extQ` combinator from `syb`.

4.4 Implementing the Worker/Wrapper Transformation

Now that we covered all of the analyses that are performed by the plugin, we can move on to the actual implementation of the worker/wrapper transformation. Unlike the presentation of the transformation scheme in Section 3.3 suggests, our implementation does not directly rewrite the lifted program. While the explicit representation of effects enables us to reason about the transformation very precisely, operating on the lifted AST would make the implementation more difficult because the involved data structures are already quite deeply nested and lifting makes them even more complicated. The approach that we have chosen instead is to first split the program into worker and wrapper functions and then lift the worker and wrapper functions in a separate phase. In this section, we focus on the first part while our extensions to the monadic lifting are covered in Section 4.5.

[5] Used to represent "multi-way if-expressions" from the **`MultiWayIf`** language extension.
[6] This also includes the "lambda case-expressions" from the **`LambdaCase`** language extension

4.4.1 Filtering Bindings

The first step is to decide whether to split a function into a worker and a wrapper at all or leave it as is. Most importantly, special bindings such as record selectors always need to be skipped. For regular function bindings, the decision to split a function must be guided by some heuristic that estimates the potential savings of the transformation. The heuristic used by the GHC takes multiple factors into account like the number of arguments, their strictness, the size of the right-hand side or whether there is an `INLINE` pragma. In our setting, however, any function with at least one argument potentially profits from the transformation. Even if the argument is not strictly evaluated, removing the `return` and `Func` around the lambda abstraction is always beneficial. Therefore, we just check the arity and split the function if it is greater than zero.

In order to check the arity, we simply count the number of arguments in the demand signature in the function and therefore do not have to analyze the AST again. However, we need to exclude dictionary arguments because they only exist in the Core representation whereas the worker/wrapper transformation operates on the Haskell AST. To tell how many arguments are dictionary arguments, we can look at the evidence binders that are attached to the function binding by the type checker.

The only kind of function that potentially does not profit from a worker/wrapper transformation is a function that is inlined anyway. While transforming such a function is redundant, the transformation does not hurt either except for increasing compilation time. To keep the implementation simple, we thus do not follow the example of the GHC and transform functions with `INLINE` pragma as well.

4.4.2 Transforming Function Bindings

As can be seen in Figure 4.1, the worker/wrapper transformation is executed after the pattern matching compiler. This allows us to make assumptions about the structure of the program which greatly simplify the transformation process. Basically, all functions we are concerned with have the following shape where x_1 to x_n are variable patterns and e is an arbitrary expression.

$$f :: A_1 \rightarrow \ldots \rightarrow A_n \rightarrow B$$
$$f = \backslash x_1 \rightarrow \ldots \rightarrow \backslash x_n \rightarrow e$$

Nevertheless, the underlying data structures are still designed to represent arbitrary Haskell programs. In consequence, navigating the AST is still quite challenging. To make things easier, we developed a series of abstractions that allow us to work with the AST in a more convenient way.

- ▷ A writer monad `WorkerWriterM` is used to collect information about all workers that are generated during the transformation of the module. Therefore, we do not have to deal with additional return values. The information that is aggregated in this way guides the modified monadic lifting of the worker functions that will be discussed in the next section.
- ▷ Another monad `WorkerWrapperM` with reader semantics is used during the transformation of a single function binding to pass information down to functions transforming the right-hand side. Therefore, we do not have to explicitly pass arguments around.

▻ Lastly, we defined a record data type `WorkerWrapper` A that acts as a container for two AST nodes of type A which resulted from the worker/wrapper transformation. The named fields of this data type prevent confusion about which node belongs to the worker and wrapper, respectively. A `Functor` instance allows us to apply a function to both nodes at the same time and an `Applicative` instance enables us in conjunction with a monadically lifted `traverse` to worker/wrapper transform a list of AST nodes (or any other `Traversable`) without any boilerplate. These mechanisms are particularly handy because a lot of AST nodes are wrapped in a type that adds source location information.

For example, the following code snipped demonstrates how we can apply the transformation to all `Match`es of a `MatchGroup`. The result is a `WorkerWrapper` of the transformed `mg_alts`. Despite the three levels of nesting that are involved not much boilerplate is required.

```
traverseM (traverseM (traverseM transformMatch)) (mg_alts matchGroup)
```

Using these abstractions, we transform a function of the shape shown above into two function bindings of the following form. Why we have to "collapse" the lambda abstractions on the right-hand side of the worker function into the left-hand side will become evident in Section 4.5.

$$f :: A_1 \to \ldots \to A_n \to B$$
$$f = \backslash x_1 \to \ldots \to \backslash x_n \to f_{\texttt{worker}}\ x_1\ \ldots\ x_n$$

$$f_{\texttt{worker}} :: A_1 \to \ldots \to A_n \to B$$
$$f_{\texttt{worker}}\ x_1\ \ldots\ x_n = e$$

The new identifier $f_{\texttt{wrapper}}$ of the worker function is generated by adding a new `Unique` to the original identifier f. For debugging purposes, we also add the suffix `_worker`. In case of symbolic names, we cannot use alphanumeric characters and therefore append `\/\/` instead. All other details are copied from the original identifier. Importantly, this includes `INLINE` pragmas such that if f is inlined, the worker $f_{\texttt{worker}}$ is also inlined.

Regardless of the pragmas set by the user, we need to ensure that the wrapper function is inlined such that the GHC can apply rewrite rules. We can use the GHC's constant `alwaysInlinePragma` to enable inlining in all phases of the compiler. Additionally, we need to specify that the GHC is allowed to inline the function when it is applied to zero arguments by setting the pragma's `inl_sat` accordingly. Unfortunately, it is not sufficient to add the pragma only to the identifier in the function definition. There is no single source of truth for information attached to identifiers. The pragma has to be set on every occurrence of the identifier in the program instead. We can do so by applying a substitution to the entire program after the transformation.

Because of the `INLINE` pragma that we add to the wrapper function's identifier, the GHC will automatically inline the wrapper into the worker's right-hand side in case of recursive functions. This is why we can deviate from the schema in Figure 3.3 and use the original right-hand side e. In consequence, the worker and wrapper function are mutually recursive, though. In Section 4.4.4 we will be confronted with the implications of this design decision.

4.4.3 Transforming Evidence

One aspect of the AST that the code listings from the previous section cannot capture is how the additional information that is attached to AST nodes by the type checker is influenced by the transformation. After type checking function bindings and expressions can have an `HsWrapper`. Conceptually, an `HsWrapper` can be understood as a small snippet of GHC Core code that needs to be wrapped around an expression when it is desugared. The wrappers are needed because type arguments and dictionaries are implicitly passed to functions whereas they are explicitly passed in Core.

The simplest `HsWrapper` is a `WpHole` which indicates that no additional code needs to be wrapped around an expression. Since function bindings need lambda abstractions for their invisible arguments, there are `WpTyLam` v and `WpEvLam` v that cause a type or evidence abstraction for a variable v to be wrapped around the expression. Analogously there exist `WpTyApp` τ and `WpEvApp` d for type and evidence applications. When multiple pieces of code need to be wrapped around an expression, the corresponding wrappers w_1 and w_2 are combined using `WpCompose` w_1 w_2 to a wrapper that first applies w_2 and then w_1.

For the worker's binding, we need to use the `HsWrapper` as the original function's binding has. For the application of the worker function, we need to add application `HsWrapper`s that exactly correspond to the lambda `HsWrapper`s. We have the following two rules to create an application from a lambda abstraction wrapper.

```
transformHsWrapper :: HsWrapper -> HsWrapper
transformHsWrapper (WpTyLam v) = WpTyApp (TyVarTy v)
transformHsWrapper (WpEvLam v) = WpEvApp (EvExpr (Var v))
```

Additionally, we have to swap the order of the `WpCompose`d wrappers because the outermost lambda abstraction is the first that needs to be applied. All other kinds of `HsWrapper`s can be safely ignored. The `<.>` operator is a smart constructor for `WpCompose` that takes `WpHole`s into account.

```
transformHsWrapper (WpCompose w1 w1) = transformHsWrapper w1 <.> transformHsWrapper w1
transformHsWrapper _ = WpHole
```

4.4.4 Transforming Generalized Bindings

In the previous subsection we have seen that the type checker adds information to the AST in the form of `HsWrapper`s that we need to preserve. Additionally, we need to be careful because the type checker can also reorganize the AST. Specifically, multiple `FunBind`s can be grouped into a `AbsBinds` node. In this subsection, we discuss how the metadata that is stored in such a node needs to be adapted, how a surrounding `AbsBinds` node changes the transformation of the contained function bindings and under which circumstances the resulting worker and wrapper functions can reside in the same `AbsBinds` node or have to be separated.

First, we need to understand why the type checker introduces such a node in the first place. Consider the following function declaration.

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

Because the binding lacks a type signature, the type checker needs to infer the type of `reverse`. To this end, it first supposes that there exists some unknown but fixed type α_0 such that `reverse ::` α_0. Through the application of a series of inference rules, the type checker eventually unifies α_0 with $[\alpha_1]$ `->` $[\alpha_2]$ for some other unknown but fixed types α_1 and α_2. This monomorphic type now needs to be generalized to a polymorphic type `forall a. [a] -> [a]`. For this purpose, the type checker splits the function into a polymorphic top-level binding and a monomorphic local binding. Using the `ScopedTypeVariables` language extension we can write down a roughly equivalent program that illustrates the type checker's output.

```
reverse :: forall a. [a] -> [a]
reverse = reverse^mono
  where
    reverse^mono :: [a] -> [a]
    reverse^mono []     = []
    reverse^mono (x:xs) = reverse^mono xs ++ [x]
```

The type variable `a` is bound by the polymorphic function and scopes over the definition of the monomorphic binding. Note how the monomorphic function, recursively calls itself and does not refer back to the polymorphic identifier, i.e., the polymorphic function is not recursive itself. In the monomorphic function's recursive call, the `HsWrapper` also does not need a type application.

While the process of generalizing over type and evidence arguments is only necessary if there is no type signature, the GHC still performs the same splitting of the function into a polymorphic and monomorphic part even if there is a type signature. In this scenario, the monomorphic function, counterintuitively, is not monomorphic at all, i.e., this time $\texttt{reverse}^{\texttt{mono}}$ has a `HsWrapper` for a type lambda whereas `reverse` does not. Furthermore, the recursive call targets the polymorphic function. Its `HsWrapper` therefore needs to apply the type argument. In the listing below, this fact is illustrated using the syntax form the `TypeApplications` language extension.

```
reverse :: [a] -> [a]
reverse = reverse^mono
  where
    reverse^mono :: forall b. [b] -> [b]
    reverse^mono []     = []
    reverse^mono (x:xs) = reverse @b xs ++ [x]
```

While functions with a type signature are also placed into `AbsBinds` in order to model this splitting into monomorphic and polymorphic functions, they are never grouped with other bindings. Functions without type signatures, on the other hand, may need to be grouped with other bindings in case they are mutually recursive. This is because the recursive calls refer to the monomorphic identifiers that are only available in the local scope. For instance, two mutually recursive functions f and g without type signatures have `AbsBinds` that correspond to the following definition.

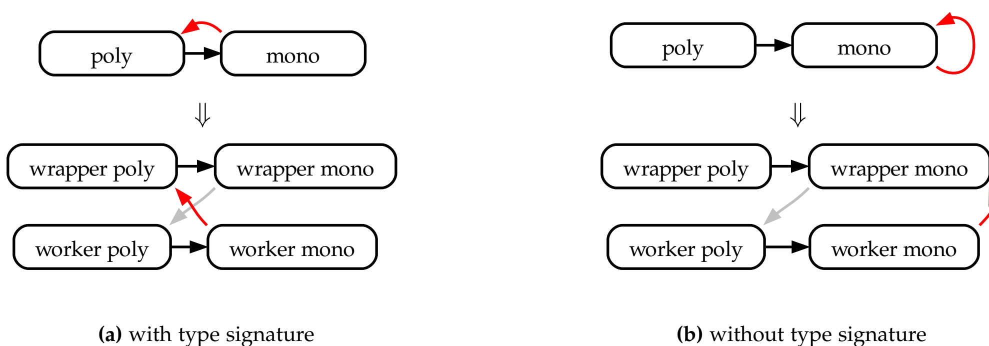
```
f, g :: A -> B
(f, g) = (f^mono, g^mono)
  where
    f^mono :: A -> B
    f^mono = body_f g^mono

    g^mono :: A -> B
    g^mono = body_g f^mono
```

Now that we familiarized ourselves with the reasons and circumstances under which the GHC's type checker creates `AbsBinds` nodes, we can apply this knowledge to determine how we need to adapt our transformation in order to produce `AbsBinds` that the GHC can compile. The following questions need to be answered in this regard.

1. When the wrapper function invokes the worker, do we need to use the monomorphic or polymorphic identifier?
2. Do the worker and wrapper functions need to be in the same `AbsBinds` node or do we have to split them apart?

These two questions are linked. The answer to one constraints how the other can be answered. For example, if we decide to put the worker and wrapper functions into separate `AbsBinds` nodes, they cannot refer to each other using the monomorphic identifier because it would not be in scope. Conversely, if we decide to use the monomorphic identifiers, we must put the worker and wrapper functions into the same `AbsBinds`. Thus, we choose to always refer to the worker function using the polymorphic identifier for now.

To answer the second question, we need to compare the call graphs between the monomorphic and polymorphic functions before and after worker/wrapper transformation as shown in Figure 4.2. We only consider a single recursive function binding for simplicity but need to distinguish whether the function has a type signature or not.



(a) with type signature

(b) without type signature

Figure 4.2. Call graphs of a recursive function with and without type signature. The corresponding call graph of the worker/wrapper transformed function is shown at the bottom. Black arrows indicate the dependency between polymorphic and monomorphic functions. The recursive calls are shown in red and the call of the worker function from the wrapper is shown in gray.

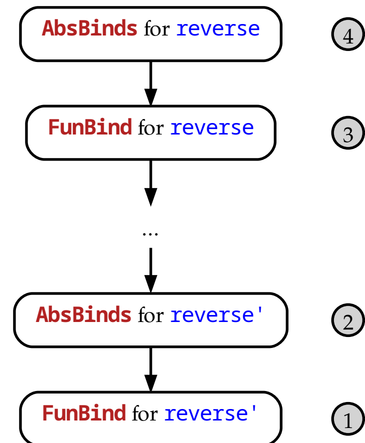
As we noted at the end of Section 4.4.2 already, we can see here that the worker and wrapper functions are indeed mutually recursive. The red arrows in Figure 4.2 indicate the original recursive calls. Since the worker function's right-hand side is exactly the original right-hand side, the recursive calls in the worker function target the same callee as before, i.e., the polymorphic identifier if there is a type signature and the monomorphic identifier otherwise. In the latter case, the two functions therefore cannot be separated. In the other case, they actually must be separated, because the GHC expects there to be only one function in `AbsBinds` when there is a type signature.

To tell whether we must split an `AbsBinds` node, we can simply look for recursive calls to the monomorphic identifier in the original bindings. If there is any such call, we cannot split the `AbsBinds` node after worker/wrapper transforming the contained function bindings. The remaining transformation of `AbsBinds` involves updating the association between polymorphic and monomorphic identifiers and ensuring that the call to the polymorphic worker function is wrapped with appropriate `HsWrappers` for generalized type and evidence arguments. With the transformation of `AbsBinds` in place, we can now worker/wrapper transform arbitrary top-level bindings.

4.4.5 Transforming Local Bindings

In addition to top-level bindings we also want local bindings to be transformed. Generally, we can use the same approach as for top-level bindings. However, we need a mechanism to find all of the local bindings in the first place. As per usual, the `syb` library is our preferred choice for this task because it allows us to traverse the AST in a generic way. Unfortunately, we cannot just use `everywhereM` since this would cause function bindings to be processed twice. To understand why this is the case, consider the following example function. The shape of the AST after type checking is illustrated on the right where the numbers indicate the order in which the nodes would be visited by `everywhereM`.

```
reverse :: [a] -> [a]
reverse = reverse' []
where
  reverse' :: [a] -> [a] -> [a]
  reverse' acc []     = acc
  reverse' acc (x:xs) = reverse' (x:acc) xs
```



Since `everywhereM` processes nodes in a bottom-up fashion, the function binding for `reverse'` is processed first. Since we cannot see the surrounding `AbsBinds` at this point, we split the function like an unabstracted top-level binding. Next, the traversal goes one level up to visit the `AbsBinds`. Splitting the `AbsBinds` involves splitting the contained function bindings. Therefore, we split the worker and wrapper that were just created once again. The same problem occurs for `reverse` and its `AbsBinds` on

the way up.

Alternatively, we need a version of `everywhereM` that visits nodes in top-down order and stops as soon as a matching node has been found such that we can explicitly specify where to continue the traversal. The `syb` library provides a function `somewhere` that does exactly that. This function only descends into the children of a node if the given monadic transformation fails by the means of `mzero` from the `MonadPlus` type class.

```
somewhere :: forall m. MonadPlus m => GenericM m -> GenericM m
```

Using `somewhere` we define the following function to apply the transformation to all local bindings in a given AST node. This function needs to be invoked recursively on the transformed function bindings to process arbitrarily nested local bindings.

```
transformLocalBinds :: (Data a) => DemandMap -> a -> WorkerWriterM a
transformLocalBinds demandMap x =
  fmap (fromMaybe x) $
    runMaybeT $
      somewhere (extM (const mzero) (lift . transformBindsBag demandMap)) x
```

The function first `lift`s a computation from our transformation's writer monad into the `MaybeT` monad transformer. We need `MaybeT` such that we get an instance of `MonadPlus`. Then we convert the lifted function into a generic monadic transformation but instead of `mkM` we use `extM (const mzero)`. This means that the transformation will fail for all nodes that are of a different type than expected by `transformBindsBag`. In consequence, this definition allows us to descend until we find a node of the expected type and then stop. After we apply the transformation `somewhere` in the AST node `x`, we no longer need the `MaybeT` transformer. In case the entire computation failed, i.e., there are no local bindings, we finally return the original AST node `x` instead of `Nothing`.

4.5 Monadic Lifting of Workers and Wrappers

In the previous section, we have seen how we split function bindings into a worker and wrapper function. Both functions still need to be lifted. In this section, we discuss how we had to modify the plugin's existing lifting to add support for the translation of bindings and applications of worker functions.

4.5.1 Application Schemas

To lift the worker and wrapper functions, we want to reuse as much as possible from the original lifting. However, the lifting for bindings and applications of worker functions slightly differs from the regular lifting. Namely, the function itself is not wrapped in the monad and strict arguments are neither. Thus, we need additional information to guide the insertion of `app`, `(>>=)`, and `return` in

the lifting process. During worker/wrapper transformation we therefore collect so-called *application schemas* for each worker function. Application schemas are metadata that describe how the function is intended to be invoked after lifting but are also used to steer the lifting of the function binding's right-hand side. These application schemas are represented by the following data type.

```
data AppSchema
  = FuncAppSchema
  | LazyAppSchema !AppSchema
  | StrictAppSchema !AppSchema
```

The `FuncAppSchema` constructor indicates that that the regular lifting should be used for the application of the function, i.e., the plugin needs to use the `app` operator. The other two constructors deal with the application of functions that are not wrapped into the monad. While a `LazyAppSchema` indicates that the argument is still wrapped in the monad, a bind needs to be inserted for the argument before the function application if a `StrictAppSchema` is encountered. Both constructors take an `AppSchema` as argument. This residual application schema describes how the result of the function needs to be applied. The result of a function with `FuncAppSchema` always needs to be applied using a `FuncAppSchema` as well. During lifting, we can assume that we eventually get a `FuncAppSchema` as the residual application schema because the worker/wrapper transformation is guaranteed to fully apply the worker function.

For presentational purposes, we introduce the following notation for the i-th residual application schema.

$$
\begin{array}{rcll}
\forall a :: \texttt{AppSchema}: a_i & = & a & \text{if } i = 0 \\
\texttt{FuncAppSchema}_i & = & \texttt{FuncAppSchema} & \text{if } i > 0 \\
\forall a :: \texttt{AppSchema}: (\texttt{LazyAppSchema}\ a)_i & = & a_{i-1} & \text{if } i > 0 \\
\forall a :: \texttt{AppSchema}: (\texttt{StrictAppSchema}\ a)_i & = & a_{i-1} & \text{if } i > 0
\end{array}
$$

4.5.2 Lifting Types

Before we can start lifting the actual code, we need to understand how an application schema influences the lifting of a function's type. If we denote the lifting of a function type using an application schema a `::` `AppSchema` as $[\![\tau]\!]^{fun}_a$, the modified lifting of types is governed by the following rules.

$$
\begin{array}{rcl}
[\![\tau]\!]^{fun}_{\texttt{FuncAppSchema}} & = & [\![\tau]\!] \\
\forall a :: \texttt{AppSchema}: [\![\tau \texttt{ -> } \tau']\!]^{fun}_{\texttt{LazyAppSchema}\ a} & = & [\![\tau]\!] \texttt{ -> } [\![\tau']\!]^{fun}_a \\
\forall a :: \texttt{AppSchema}: [\![\tau \texttt{ -> } \tau']\!]^{fun}_{\texttt{StrictAppSchema}\ a} & = & [\![\tau]\!]^i \texttt{ -> } [\![\tau']\!]^{fun}_a
\end{array}
$$

In case of a `FuncAppSchema`, the regular lifting is applied to the function. For `LazyAppSchema`s the function arrow is not lifted. The same is true for `StrictAppSchema`s but we additionally use the inner lifting for the parameter type to prevent it from being wrapped in the monad.

Note, that if $\tau = $ `forall a.` σ for some type σ, there is no monad to be removed on the outermost level in the lifted type, i.e., the following equation holds.

$$\begin{aligned} [\![\tau]\!]^i &= [\![\texttt{forall a. } \sigma]\!]^i \\ &= \texttt{forall a. } [\![\sigma]\!] \\ &= [\![\tau]\!] \end{aligned}$$

A situation like this can occur if the `RankNTypes` language extension is enabled. The absence of the monad implies that if a function has a strict polymorphic argument, we cannot bind this argument using `(>>=)`. Therefore, we need to be careful when we lift such a function or an application thereof not to bind or `return` the polymorphic argument.

4.5.3 Lifting Applications

In order to lift a worker function application, we first need to collect all arguments e_1 through e_n an expression f is applied to. If f is a variable expression, we look up its application schema a `::` `AppSchema`. In case f is not a worker function we set $a =$ `FuncAppSchema`. The following two aspects of the lifting now need to be modified.

1. We only insert `app` for the application of e_i if the corresponding application schema a_{i-1} is a `FuncAppSchema`.
2. If $a_{i-1} =$ `StrictAppSchema` a_i, the entire application expression needs to be wrapped in a bind for e_i. Note that we cannot insert the bind in the middle of an application because the intermediary results are not lifted into the monad.

For this purpose, we define an auxiliary lifting $[\![\cdot]\!]^{app}$ that lifts some application expression. The lifting produces a tuple where the first component is an expression that contains a placeholder $\bullet$ that finally needs to be substituted with the expression in the second component. Conceptually, the second component is the lifted application expression and the first component gathers binds that eventually need to be wrapped around the resulting expression. Additionally, a third component records the residual application schema. The substitution of the placeholder symbol is illustrated by the equation below.

$$[\![f\ e_1\ \ldots\ e_n]\!]^e = w[r/\bullet] \quad \text{if } [\![f\ e_1\ \ldots\ e_n]\!]^{app} = (w, r, \texttt{FuncAppSchema}) \text{ and } n > 0$$

At this point, the residual application schema must be `FuncAppSchema` because the wrapper always applies the worker to the correct number of arguments. This also guarantees that r is wrapped in the monad such that the replacement is type correct. In the actual implementation, the replacement is realized by representing the first component of the tuple as a function that maps an `HsExpr` to a new `HsExpr`.

The auxiliary lifting $[\![\cdot]\!]^{app}$ is inductively defined. In the base case, the function f is applied to no arguments. The regular lifting can be applied in this case.

$$[\![f]\!]^{app} = (\bullet, [\![f]\!]^{e}, a) \quad \text{where } a \texttt{::} \ \texttt{AppSchema} \text{ application schema of } f$$

Given an expression e with $[\![e]\!]^{app} = (w, r, a)$, the lifting of an application $e\ e'$ depends on the residual application schema a of e.

- ▻ If $a =$ `FuncAppSchema`, then r is wrapped in the monad and `app` must be used for the application.
$$[\![e\ e']\!]^{app} = (w,\ r\ \texttt{`app`}\ [\![e']\!]^{e},\ \texttt{FuncAppSchema})$$
- ▻ If $a =$ `LazyAppSchema` a', then r is not wrapped into the monad and can be applied directly.
$$[\![e\ e']\!]^{app} = (w,\ r\ [\![e']\!]^{e},\ a')$$
- ▻ If $a =$ `StrictAppSchema` a', then r can also be applied directly, but its argument needs to be bound first.
$$[\![e\ e']\!]^{app} = ([\![e']\!]^{e} \texttt{ >>= } \backslash x \texttt{ -> } w,\ r\ x,\ a') \quad \text{where } x \text{ fresh variable}$$

While these rules are already complicated on their own, the real implementation is even more complex because we also need to make sure that the resulting expression have the correct type information attached. We also need to handle infix operations explicitly. In principal, the equations fully capture the plugin's behavior, though.

4.5.4 Lifting Function Bindings

The lifting of a worker function's binding is a little bit easier because of a tick that we applied in Section 4.4.2. Instead of writing the arguments of the worker function into lambda abstractions on the right-hand side of the binding, we collected all argument patterns and wrote them on the left-hand side instead. Usually, the plugin assumes that all arguments are bound by single-argument lambda abstractions. The translation of such lambdas is usually governed by the following rule.

$$[\![\backslash x \texttt{ -> } e]\!]^{e} = \texttt{return \$ Func \$ } \backslash x \texttt{ -> } [\![e]\!]^{e}$$

This rule is responsible for the insertion of the `return` and `Func` but only applies to lambda abstractions. In our case, this has the fortunate effect that `return` and `Func` are not inserted for the worker function's binding. Nevertheless, since function bindings and lambda abstractions both use `MatchGroup`s internally to represent their pattern matching and right-hand sides, we can still reuse the rest of the plugin's lambda abstraction lifting logic for lifting our worker functions.

The only remaining problem is that we need to generate local bindings that `return` the strict arguments of the worker function. For this purpose, we slightly modified the lifting of `Pat`terns and `Match`es. The function that is responsible for lifting patterns is also responsible for generating fresh identifiers for shared variables. Basically, a variable pattern x is replaced by a fresh variable pattern x'.

The plugin then creates a `share` x' `>>= \x ->` ... on the right-hand side of the pattern's corresponding `Match`. Analogously, we replace the variable pattern of a strict argument with a fresh variable pattern x''. The plugin then creates `let` x' `= return` x'' `in` ... on the right-hand side of the pattern's corresponding `Match` right before the `share`.

With this little change, we are already done with our modifications to the plugin's lifting for the regular worker/wrapper transformation. In the final subsection, we will discuss which additional changes have to be made to the lifting processes in order to support the `unsafeBind` optimization presented in Section 3.3.4.

4.5.5 Extended Lifting using `unsafeBind`

While the effect analysis currently does not suffice to prove that the usage of `unsafeBind` is safe, we still want to implement experimental support for this extended transformation such that we can evaluate in the next chapter whether this optimization has the expected effect on the performance of the generated code. Instead of using the result of effect analysis, we added the following annotation data type. A user can add a `EnableExperimentalUnsafeBind` annotation to a function binding to indicate that the plugin should use `unsafeBind` when lifting its worker function.

```
data ExperimentalAnnotation = EnableExperimentalUnsafeBind
  deriving Data
```

In this subsection, we describe the changes that we had to make to the plugin's lifting to support this experimental feature. First, the application schema data type had to be extended such that we can express that the result of a function application is not lifted. Therefore, we added a flag to `FuncAppSchema` that indicates whether the `Func` itself is wrapped in the monad or not.

```
data LiftedFlag = IsLifted | IsNotLifted
data AppSchema
  = FuncAppSchema LiftedFlag
  | ... -- Remaining constructors unchanged.
```

If the `EnableExperimentalUnsafeBind` annotation is set for the original function, the worker/wrapper transformation sets the `LiftedFlag` of the residual `FuncAppSchema` to `IsNotLifted`. Otherwise, the flag is set to `IsLifted`. All rules from the previous sections that applied to `FuncAppSchema` still apply to `FuncAppSchema IsLifted`. For `FuncAppSchema IsNotLifted`, the following additional rules need to be defined.

1. If this kind of function were applied, the result would be lifted into the monad.

$$(\texttt{FuncAppSchema IsNotLifted})_i = \texttt{FuncAppSchema IsLifted} \quad \text{if } i > 0$$

2. If the final application schema during lifting of a type indicates that the result is not lifted, we need to lift the entire result type with the inner lifted to prevent the monad from being wrapped around.

$$[\![\, \tau \,]\!]^{fun}_{\texttt{FuncAppSchema IsNotLifted}} = [\![\, \tau \,]\!]^{i}$$

3. If the final application schema in an application indicates that the result is not lifted, we need to insert a `return`, i.e., if $[\![\, f\ e_1\ \dots\ e_n \,]\!]^{app} = (w, r, \texttt{FuncAppSchema IsNotLifted})$ and $n > 0$, then the application expression is lifted as follows.

$$[\![\, f\ e_1\ \dots\ e_n \,]\!]^{e} = w[\texttt{return}\ r/\bullet]$$

4. Since the worker function is never applied to too many arguments, we do not need a rule for $[\![\, e\ e' \,]\!]^{app}$ that covers the case that the residual application schema returned by $[\![\, e \,]\!]^{app}$ is **`FuncAppSchema IsNotLifted`**.

5. Finally, the entire worker function's right-hand side is wrapped in `unsafeBind` if a is the worker's application schema, n its arity and $a_n = \texttt{FuncAppSchema IsNotLifted}$.

The performance benefits of this experimental extension are evaluated in the next chapter along with the evaluation of our regular optimization.

Chapter 5

Evaluation

In this chapter we evaluate whether the implementation of the optimization described in the previous chapter has the desired effects. In the first section we present our testing approach that we have used to ensure that the optimization does not change the semantics of the program or cause any runtime errors. Our tests also verify that the demand analysis produces the expected demand signatures as presented in Section 5.2. In the third section we present the results of benchmarks that we conducted to evaluate the impact of the optimizations on the runtime performance, memory consumption and size of generated code as well as the increase of compilation time that the optimizations entail. Finally, we consider the impact our changes have on the plugin's maintainability. Throughout this chapter we focus on the Curry GHC Language Plugin. However, the results of our evaluation are expected to apply to other instantiations of the plugin as well.

5.1 Automating Existing Tests

To ensure that our modifications to the plugin do not break existing functionality or change the semantics of the translated programs, we need to test the code produced by the plugin. Fortunately, the original implementation already came with multiple example modules, that we can use for testing. However, most of these examples were not executed automatically by the test suite but only checked for compile-time errors. Furthermore, the tests for those examples that were actually executed by the test suite have not been stated in a scalable manner due to the use of a custom test framework. In this section, we present how we rewrote and automated the plugin's test suite to improve our confidence in the correctness of our implementation. First, we introduce Hspec[1] a popular framework for testing Haskell programs. Then, we outline how we use Hspec in the new test suite and finally present our findings of executing the resulting tests before and after our implementation from Chapter 4.

5.1.1 Hspec: A Testing Framework for Haskell

To implement our test suite we have chosen the testing framework Hspec. The framework defines a Domain Specific Language (DSL) for writing tests in Haskell. The DSL is based on the `Spec` monad which is mainly built around the following functions provided by Hspec.

```
describe :: String -> Spec -> Spec
it :: String -> Expectation -> Spec
```

[1] https://hspec.github.io/

The `describe` function is used to define a group of test cases. The first argument is a description of the group and the second argument is a specification of the group which is itself comprised of `describe` and `it` items. The `it` function is used to define a single test case. The first argument is a description of the test case and the second argument is the expectation that must be fulfilled for the test case to pass. An `Expectation` is basically just a computation in the `IO` monad, that throws an exception when the expectation is not fulfilled. This fact is mostly transparent to the user as Hspec provides its own operations for defining expectations. Of the many operations that are available, we will only need the following three for our test suite.

```
shouldBe :: (HasCallStack, Show a, Eq a) => a -> a -> Expectation
expectationFailure :: HasCallStack => String -> Expectation
pendingWith :: HasCallStack => String -> Expectation
```

The `shouldBe` operator takes two arguments and sets the expectation that the two arguments are equal. The `expectationFailure` function creates an expectation that always fails. The given string contains the reason for the failure. Finally, the `pendingWith` function creates an expectation that is always skipped by Hspec. The given string contains the reason for skipping the expectation. This function is useful for temporarily disabling tests that are known to fail.

A key reason for our choice of Hspec is its discovery mechanism. The old test suite of the plugin required all test cases to be added to a `tests` list exported by the main module. When adding a new test case, we had to remember to add it to this list. This approach is not scalable and reduces the confidence in the test results since we never know for sure that we have not forgotten to add a test case to the list. There is also the risk, that a test case has been added to the list but while resolving a Git merge conflict, the corresponding line has been removed on accident. When using Hspec, all of our `Spec` items also have to be collected in the main module and passed to the `hspec` function to check the contained expectations. However, we do not have to do this ourselves since the framework provides a GHC plugin that automatically discovers all test cases in a project and generates the corresponding `main` function. Hspec even inserts `describe` calls to make it easier to identify the module that a test case originated from.

5.1.2 Converting Exiting Examples to Hspec

To convert the existing examples to tests that can be checked by Hspec, we first moved the example modules from the `test-examples` directory into the `test` source directory. Since the examples are now part of the test suite's source code, they are compiled automatically when the test suite is built. Thus, we eliminated the need to compile the examples manually by spawning GHC processes at runtime. Since we also converted the Cabal files to the Hpack[2] format, we don't even have to add the module names to the built configuration. To add a test, it is sufficient to add the file to the `test` directory.

The approach above works perfectly for all examples that are expected to compile successfully. However, there is one example, that is expected to fail. To handle this edge case, we can add an exception to the test suite's built configuration in `package/curry-ghc-language-plugin/package.yaml`.

[2] https://github.com/sol/hpack

```yaml
tests:
  hspec:
    # ...
    when:
      - condition: false
        other-modules:
          - Example.HaskellImport
```

This configuration causes the `Example.HaskellImport` module whose compilation we expect to fail to be built only when the condition `false` is satisfied, i.e., never. Next, we add a corresponding `Example.HaskellImportSpec` module that exports a single `Spec` item that runs the GHC on the excluded module similar to how the old test suite compiled all of the examples. Spawning the process is possible in HSpec since `Expectation`s are just `IO` computations. When the process succeeds we use `expectationFailure` to indicate that the test failed. Otherwise, we just return which marks the test as passed.

Finally, we just have to add `Spec` modules for all other examples. Since the code produced by the plugin seamlessly integrates with Haskell code, we can simply call the example functions, encapsulate their results and check whether the results match the expected outcome by using Hspec's `shouldBe` operator. The tests that were already executable in the old test suite can be converted to Hspec tests by following the same approach.

For instance, there is the following example that demonstrates the use of Curry's `(?)` operator.

```haskell
{-# OPTIONS_GHC -fplugin Plugin.CurryPlugin #-}
module Example.Coin where

coin :: Bool
coin = True ? False
```

To test this example, we only need to add a module `Example.CoinSpec` that exports the `Spec` item depicted below.

```haskell
spec :: Spec
spec = describe "coin" $ do
  it "chooses between True and False" $ do
    eval DFS coin `shouldBe` [True, False]
```

5.1.3 Broken Tests

When we tried to run the new test suite for the first time, we found that some of the tests failed to compile, even though we haven't changed the plugin's implementation yet. The failures likely were the result of regressions introduced by a recent upgrade of the supported GHC version to 9.2.1. The original implementation of the plugin as presented by Prott (2020) targeted version 8.10.1 of the compiler. The following test cases are affected:

- ▻ `Typeclass`
  In this example module a custom type class and instances thereof are defined.
- ▻ `MultiParamFlexible`
  This example demonstrates the usage of multi-parameter type classes and other type class related language extensions.
- ▻ `DefaultSignaturesImport` and `DefaultSignaturesExport`
  In one of the example modules, a class with `default` signature is defined. The other module imports that class. Both modules define an instance of the class.
- ▻ `TypeOperatorsImport` and `TypeOperatorsExport`
  In these modules, the `TypeOperators` language extension is used. A data type with a symbolic identifier is defined in one module and used in the other module. However, there are type errors in the imported module.

Most of the failing tests are related to the declaration of new type classes in the transformed module. Examples that define instances of type classes that are built into the plugin compile successfully. We already covered the limitations of the plugin regarding type classes in Section 2.3.4. Since fixing these errors is out of scope for this thesis, we decided to exclude the affected tests from the test suite. The broken examples, the corresponding test specification and error messages emitted by the compiler can be found in the `package/curry-ghc-language-plugin/broken-test` directory.

After excluding the broken tests, we were able to compile the test suite successfully. However, there are still two tests that fail at runtime with a segmentation fault. Again, both failing tests are related to type classes. One example uses the `Applicative` type class. The other uses the `Monad` type class. For the same reason as above, we disabled these two tests using Hspec's `pendingWith` operation.

5.1.4 Test Results

In spite of the initial challenges encountered while setting up the test suite, we managed to successfully run the remaining tests. After the implementation of our optimizations, the previously functioning tests continue to pass reliably. The automated nature of the test suite proved very helpful throughout the development process, aiding in the early identification of numerous implementation oversights.

5.2 Testing Demand Signatures

The tests that we automated in the previous section check that the semantics of translated programs are not changed by our optimization. For our optimization to have the best possible performance improvements, we need the demand analysis to correctly identify when an argument of a function is evaluated strictly. However, when the demand analysis fails to identify an argument as strict, the optimization will not change the semantics but only the runtime characteristics of the program. Thus, our tests still pass even when the demand analysis exhibits unexpected behavior.

To identify such cases, we could manually inspect the demand signatures produced by the demand analysis and check whether they match our expectations. However, we would like this test to be automated as well to prevent future regressions. While the benchmarks that will be presented in Section 5.3 are automated tools that can hint at use cases that involve not fully optimized functions, the benchmarks do not provide a reliable mechanism to identify the exact cause of the observed performance problems. For functions that happen to not lie on a critical path of any benchmark, we cannot rely on the benchmarks to extract any information about the demand signature at all.

The approach that we have chosen to detect deviations from the expected demand signatures is based on annotations added by the user to functions they like to test. First, we had to define a new data type that can be used in the `ANN` pragmas and contains the expected demand signature. We based the new data type on the `DemandSignatureAnnotation` data type presented in Section 4.1.4.

```
newtype DemandAssertionAnnotation = ExpectedDemandSignature String
  deriving Data
```

Next, we added the new annotation to all functions in the Prelude as well as in the benchmark and test suites. Since the demands of dictionary arguments tend to be very complex and irrelevant to our transformation, we used a wildcard of the form <_> in place of demands for arguments that we are not interested in. For example, consider the following function from the plugin's Prelude.

```
elem :: (Eq a) => a -> [a] -> Bool
elem a = any (a ==)
```

This function is strict in its second argument because `any` needs to check the given predicate for every element of the list. Whether the first argument is evaluated strictly depends on the implementation of the `(==)` operation for the type `a` and cannot be known at compile-time. Therefore, we expect the demands `<L>` and `<1L>` for the first and second argument, respectively. Additionally, there is another demand for the dictionary argument that is added by the GHC during the conversion to Core for the `Eq a` constraint but we do not care about the exact demand for this argument. Thus, we add the following annotation to specify the demand signature that we expect and use <_> in place of the dictionary demand that we are not interested in.

```
{-# ANN elem (ExpectedDemandSignature "<_><L><1L>") #-}
```

To test whether the demand signatures are correct, we add a check after the demand analysis that looks up the annotation for each function in the module and compares the contained string with the actual demand signature. This comparison must be performed on the string representation since the wildcards render the annotations syntactically invalid. To implement the matching of wildcards, we first split the string representations into individual demands and then compare the demands one by one using the following function to check for equivalence.

```
matchesDemandString :: String -> String -> Bool
matchesDemandString _ "_" = True
matchesDemandString actualDemand expectedDemand = actualDemand == expectedDemand
```

In case of the `elem` function, our check fails and the plugin reports the error message below.

```
/package/curry-ghc-language-plugin/src/Plugin/CurryPlugin/Prelude/List.hs:135:1: error:
    Demand signature of elem does not match expected demand signature.
    Got <MP(MCM(L),A)><L>, but expected <_><L><1L>.
    |
135 | elem a = any (a ==)
    | ^
```

The third argument is missing a demand in the demand signature because our implementation of `elem` is partially point-free and the plugin does not eta-expand function declarations prior to demand analysis. If we update the annotation as shown below, the error is resolved.

```
{-# ANN elem (ExpectedDemandSignature "<_><L>") #-}
```

As this example demonstrates, the annotation mechanism allows us to detect when the demand analysis produces unexpected results. However, the annotation mechanism has the limitation that only top-level functions can be annotated. Thus, we cannot use this approach to check the demand signatures of functions that are defined in a **`where`** clause for example. Similarly, we cannot check the demands of type class operations directly. Nonetheless, the mechanism increased our confidence in the correctness of the demand analysis, helped us to fix some bugs and provided helpful hints for further improvements of the optimization.

5.3 Benchmarks

The objective of the thesis was to improve the performance of the code generated by the plugin. In this main part of the chapter, we want to analyze whether we achieved this goal. To this end, we have conducted a set of benchmarks to evaluate the influence of the optimizations on the runtime performance and memory consumption of the transformed code. The results of the benchmarks are presented in the first two subsections. Furthermore, we want to assess the costs of our optimizations. Therefore, we measured how the optimizations affect the size of the generated executables and the compilation time. The results of these measurements are presented in the last two subsections.

The conducted benchmarks are based on the benchmarks performed by Hanus, Prott, and Teegen (2022) to measure the performance of their implementation of a more efficient monad for nondeterministic computations. Additionally, we added variants of the benchmarks for example by changing whether integers or Peano numbers are used. The use of Peano numbers often adds an interesting perspective since their **`Sharable`** instance is non-trivial in contrast to the builtin instance for **`Int`**.

To run the benchmarks we are using the `criterion`[3] library. Similar to Hspec, Criterion also defines its own DSL for specifying benchmarks. The two most important building blocks of this DSL are the `bench` and `nf` functions.

[3] `https://hackage.haskell.org/package/criterion`

```
bench :: String -> Benchmarkable -> Benchmark
nf :: NFData b => (a -> b) -> a -> Benchmarkable
```

The `bench` function takes the name of the benchmark and a **`Benchmarkable`** item which is usually created by the `nf` function or related operations. `nf` accepts a function and an argument for the function. Criterion applies the function repeatedly to the same argument and then evaluates the result to head normal form. To force the evaluation of the result to head normal form, the **`NFData`** instance is required. Since the function is always applied to the same argument, it might seem wired at first that the function and the argument are separate parameters of `nf`. However, this is needed because Haskell's laziness would prevent repeated evaluation otherwise.

To be able to benchmark nondeterministic computations, we created an extension to the `nf` function that encapsulates effects of an unary nondeterministic function. A consequence of this definition is that the cost for converting between the deterministic and nondeterministic representations of the involved data types is included in our performance measurements.

```
nfND ::
  (NFData b2, Normalform Nondet a1 a2, Normalform Nondet b1 b2) =>
  Nondet (a1 --> b1) -> a2 -> Benchmarkable
nfND f = nf (eval1 DFS f)
```

Criterion applies a benchmarked function as often as possible in a five-second interval by default[4]. When the function takes longer than five seconds the function is still applied multiple times until Criterion is confident that the results are representative (typically ten times). Criterion then performs a linear regression based on the Ordinary Least Squares (OLS) regression model to correlate the number of iterations with the measured metrics[5]. Based on the results of the regression, the library reports an estimation for the metrics along with an asymmetric 95% confidence interval, i.e., the lower and upper bound for the deviation from the estimation point expected with a 95% confidence upon a repetition of the benchmark.

5.3.1 Runtime Performance

The first metric that we consider is the time it takes to execute the benchmarks. Table 5.1 summarizes the results for the largest problem instance of each benchmark. Across all benchmarks, the runtime performance is improved when the worker/wrapper transformation is enabled and our improved sharing analysis is used. How much each benchmark profits from the transformation varies widely. Some benchmarks such as the nondeterministic permutation sort algorithm `PermSort` are improved by just a few percent. However, the runtime of many other benchmarks is several hundred times faster after the optimization. In the most extreme case, reversing a list of integers is twelfth thousand times faster than before. On the other hand, reversing a list of Peano numbers is not nearly as fast, indicating that deep sharing prevails to be one of the major bottlenecks. Despite the performance improvements, there is still a noticeable gap between the runtime of transformed programs and the

[4] http://www.serpentine.com/criterion/tutorial.html#how-long-to-spend-measuring-data
[5] http://www.serpentine.com/criterion/tutorial.html#understanding-the-data-under-a-chart

equivalent pure Haskell programs across all benchmarks. In the following we will inspect the results for the AddNum and Tak benchmarks in more detail.

Table 5.1. Measured runtime for selected benchmarks with (*After*) and without (*Before*) worker/wrapper transformation plus improved sharing analysis enabled, respectively. The factor by which runtime has been reduced with respect to the unoptimized version is shown in the column labeled *Improvement*. For reference, the *Pure* column shows the runtime for an equivalent unlifted program if applicable.

Benchmark	Before	After	Pure	Improvement
`AddNumInt10`	59.19 μs	25.25 μs	—	2.34 × faster
`AddNumPeano10`	9.43 s	43.38 ms	—	217 × faster
`NaiveReverseIntList4096`	17.47 s	70.40 ms	64.00 ms	248 × faster
`NaiveReversePeanoList4096`	21.39 s	3.55 s	63.60 ms	6.03 × faster
`PermSort12`	602.90 ms	629.89 ms	—	1.04 × slower
`Primes`	102.10 ms	4.77 ms	—	21.4 × faster
`QueensInt9`	1.90 s	1.72 s	6.81 ms	1.11 × faster
`QueensPeano9`	12.96 s	11.63 s	20.47 ms	1.11 × faster
`ReverseIntList4096`	4.86 s	381.19 μs	30.02 μs	12800 × faster
`ReversePeanoList4096`	8.94 s	3.4 s	25.57 μs	2.63 × faster
`Select100`	171.58 ms	4.84 ms	—	35.4 × faster
`TakInt27_16_8`	10.45 s	33.36 ms	16.71 ms	313 × faster
`TakPeano24_16_8`	40.93 s	35.89 s	51.84 ms	1.14 × faster

AddNum is a class of benchmarks that generate an arbitrary number nondeterministically and then add the number to itself repeatedly. The differences are in the number of occurrences of the variable that holds the generated number (1, 2, 3, 4, 5 or 12)[6] and the representation of the numbers (integers or Peano numbers). The benchmark for one variable occurrence corresponds to the degenerate case and performs no additions at all but simply returns the generated number. The following code snippet shows the implementation of the `AddNum` benchmark for integers with one addition (i.e., two variable occurrences).

```
someNum_Int :: Int -> Int
someNum_Int n = if n <= 0 then 0 else (n ? someNum_Int (n - 1))

addSomeNum2_Int :: Int -> Int
addSomeNum2_Int n = let x = someNum_Int n in x + x
```

The `someNum`$_{\texttt{Int}}$ function is responsible for the nondeterministic generation of the number. The generated number is added to itself by `addSomeNum2`$_{\texttt{Int}}$. The local variable `x` that holds the generated number is shared because it occurs twice in the expression. Let m denote the number of variable occurrences. If run-time choice were used, each occurrence of the variable would duplicate the enumeration of the numbers yielding a runtime of $\mathcal{O}(n^m)$. However, since call-time choice semantics are used, the choices are not duplicated such that the algorithm produces n results and has a runtime linear in the number of choices and additions, i.e., $\mathcal{O}(n+m)$. Since we only consider $n = 100$, we expect the runtime to be linear in m. In Figure 5.1 we plotted runtime against the number of variable occurrences.

[6] Counterintuitively, the benchmark corresponding to 12 occurrences is called `addSomeNum10` and not `addSomeNum12` as the naming convention would suggest. We are sticking with the names assigned by Hanus, Prott, and Teegen (2022), though.

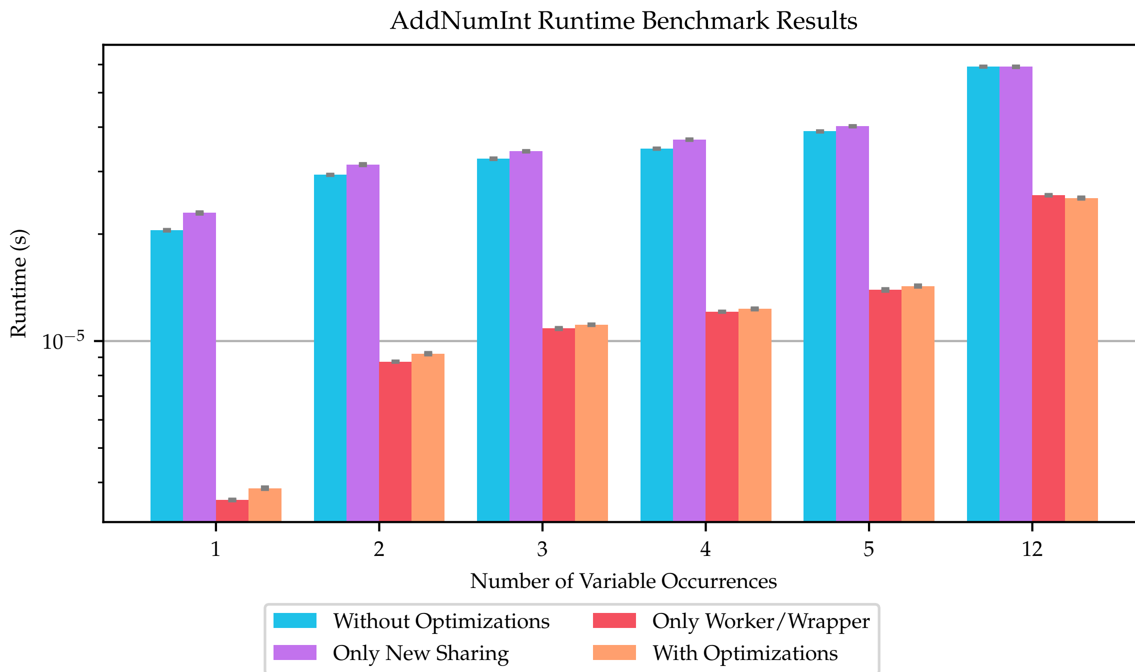


Figure 5.1. The runtime performance of the `addSomeNum` benchmarks for integers.

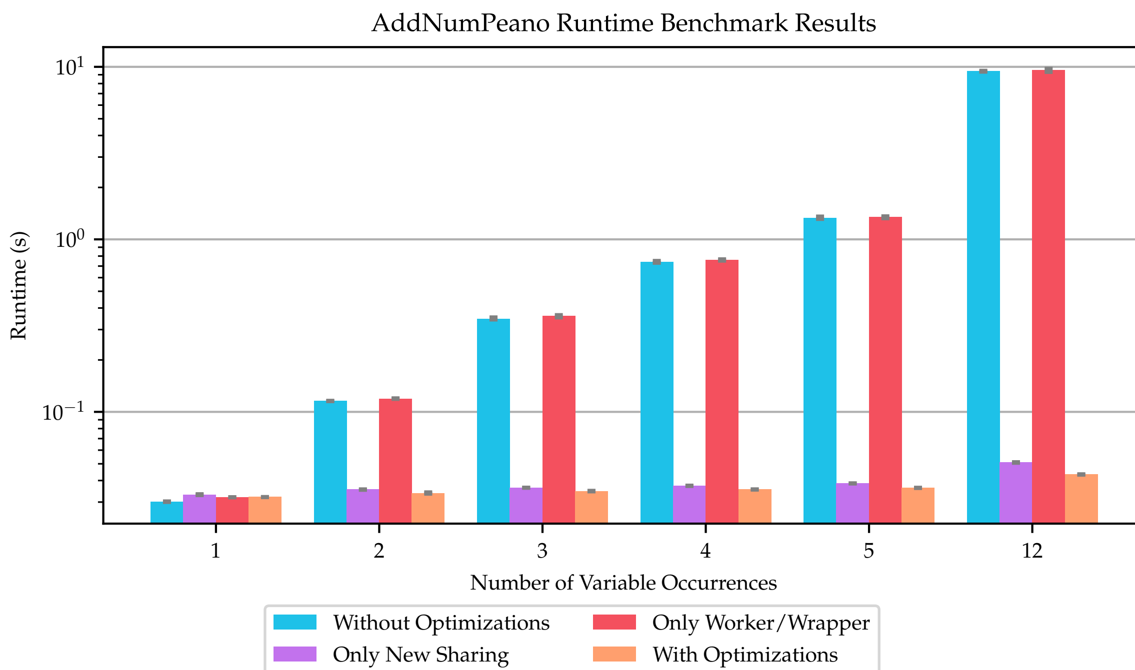


Figure 5.2. The runtime performance of the `addSomeNum` benchmarks for Peano numbers.

As expected, the runtime increases only slightly with the number of variable occurrences. While the overall runtime is already good, the worker/wrapper transformation is able to improve it by what seems to be a constant amount. Proportionately, the degenerate benchmark with $m = 1$ profits the most. This is because most of the performance gains stem from the optimization of `someNum`$_{\texttt{Int}}$. Since this function is strict in its argument, the worker/wrapper transformation helps to speed up the enumeration of the numbers. The sharing analysis does not yield any improvements for this benchmark. This is because the sharing of the duplicated `xs` is actually needed and cannot be optimized away.

A different picture is painted when we look at the Peano version of the benchmark. Let us first consider the theoretical runtime again. Below are the relevant bindings.

```
someNumNat :: Nat -> Nat
someNumNat O = O
someNumNat n@(S n') = n ? someNumNat n'

addSomeNum2Nat :: Nat -> Nat
addSomeNum2Nat n = let x = someNumNat n in x `add` x

add :: Nat -> Nat -> Nat
add O y = y
add (S x) y = S (add x y)
```

The `someNum`$_{\texttt{Nat}}$ function has the same runtime characteristics as `someNum`$_{\texttt{Int}}$. However, while `(+)` is a constant time operation, the `add` function is linear in its first argument. Thus, the overall runtime is quadratic in n for `addSomeNum2`$_{\texttt{Nat}}$. This also holds true in general. For m additions the expected runtime is $\mathcal{O}(m \cdot n^2)$ and therefore still linear in m.

Despite our expectation of linear behavior, we can clearly see exponential growth in Figure 5.2 prior to optimization. Even worker/wrapper transformation does not yield any performance improvements. However, the new sharing analysis immediately solves the problem and the runtime becomes linear[7] in the number of variable occurrences. When combining the worker/wrapper and sharing analysis, an additional slight improvement can be observed.

The performance problems are caused by the `add` function. After pattern matching compilation, the function looks as shown below. Most notably, the parameter `y` occurs twice on the right-hand side of the lambda abstraction. In consequence, the old sharing analysis concludes that a `share` needs to be inserted for `y`. Due to deep sharing, the inserted `share` has a runtime of $\mathcal{O}(n)$. In consequence, the overall runtime of `add` is increased to $\mathcal{O}(n^2)$.

```
add :: Nat -> Nat -> Nat
add = \x -> \y ->
  case x of
    O -> y
    S x' -> S (add x' y)
```

[7] In the plot the runtime appears to be constant due to a very shallow slope and the logarithmic scale of the y-axis.

However, it is not needed to share `y` because the two occurrences are in different branches of the **case** expression. Consequently, the new sharing analysis does not insert a `share`. Furthermore, `add` is strict in its first argument and can therefore be optimized by the worker/wrapper transformation. The gains from this optimization are overshadowed by the performance penalty of `share` and thus invisible when the worker/wrapper transformation is enabled in isolation.

Tak is a highly recursive function originally devised by Ikuo Takeuchi in 1978 to benchmark LISP systems (McCarthy, 1979). The function definition as shown below deviates slightly from the original definition. Takeuchi's version as remembered by McCarthy from their personal communication returns `y` instead of `z` in the **then** branch. The variant returning `z` is commonly used today (Johnson-Davies, 1986) and is actually more difficult to compute because otherwise `z` is not necessarily demanded. By returning `z` the function is strict in all three arguments meaning that all recursive calls have to be evaluated.

```
takInt :: Int -> Int -> Int -> Int
takInt x y z =
  if x <= y
    then z
    else takInt (takInt (x - 1) y z)
                (takInt (y - 1) z x)
                (takInt (z - 1) x y)
```

Unlike the functions covered in the section about `AddNum` benchmarks, `tak` is deterministic and therefore can be evaluated without lifting it with the plugin first. The runtime of the unlifted variant is shown along with the lifted function's performance in Figure 5.3. The plot demonstrates that the lifted function is slower by two orders of magnitude. The new sharing analysis does not yield any improvement since the all parameters really occur multiple times. However, worker/wrapper transformation is able to eliminate the overhead almost entirely since the function is strict. The remaining difference between the lifted and unlifted version's runtime can be explained by the monadic nature of the worker's return value.

```
takND_Int,worker :: Int -> Int -> Int -> Nondet Int
```

Since we know that neither `(<=)` from the **Ord** instance for **Int** has an effect nor any argument of the worker can introduce an effect since they have been evaluated already, we can safely add an **EnableExperimentalUnsafeBind** annotation to direct the plugin to eliminate the monad from the return type using `unsafeBind`. As the yellow bar in Figure 5.3 demonstrates, we achieve the same runtime as the unlifted variant with this optimization.

Again, it is a different story if we consider Peano numbers instead of integers. This time, the main reason is that the equivalent of the `(<=)` operator for Peano numbers `leq` is not strict in its second argument. Since zero is the smallest natural number, zero is less than or equal to any other natural number. Therefore, the second argument does not have to be evaluated when the first argument is zero. In consequence, the Peano version of `tak` is not strict in `y` either.

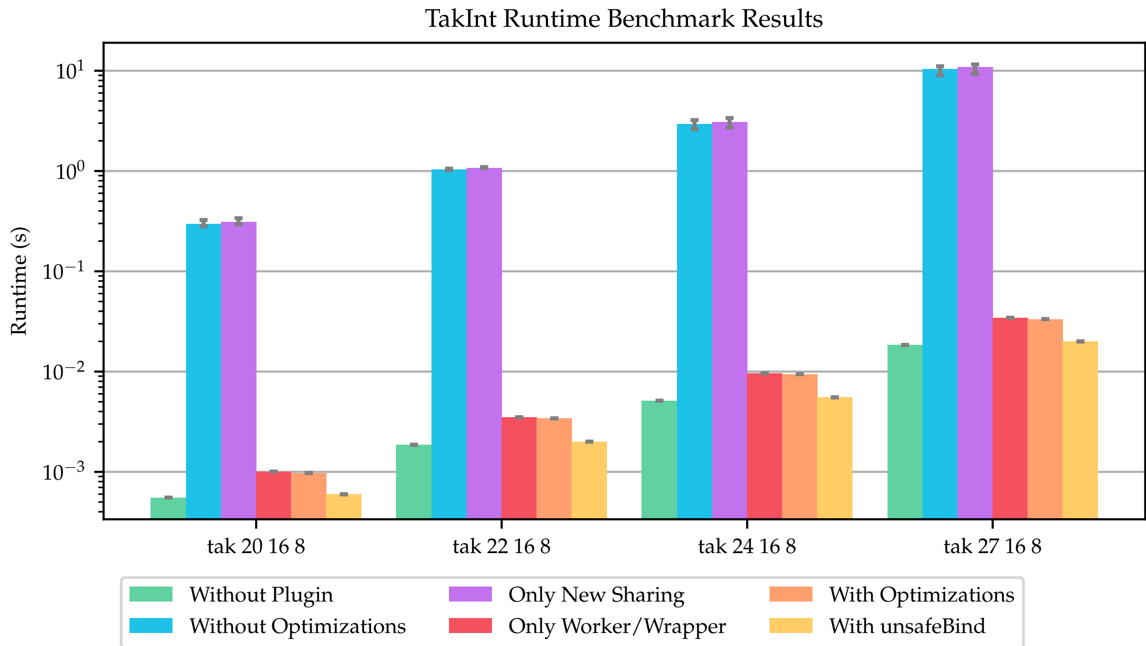


Figure 5.3. The runtime performance of the `tak` benchmarks for integers.

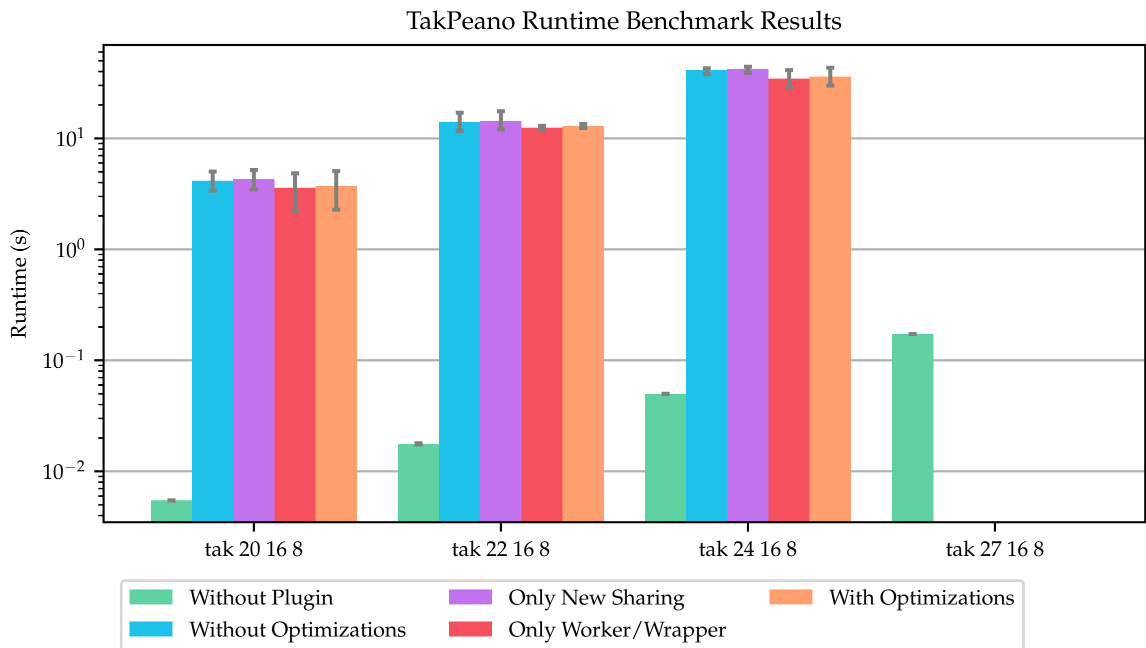


Figure 5.4. The runtime performance of the `tak` benchmarks for Peano numbers.

As Figure 5.4 shows, the runtime of the lifted function is still improved through worker/wrapper transformation but only by an insignificant amount. In fact the general performance of the generated code is so bad that the last benchmark in the series cannot be executed on our test machine even with both optimizations enabled. Furthermore, we cannot enable the experimental annotation for `unsafeBind` this time because `y` can have an effect and the predecessors of `S` constructors can also introduce effects.

5.3.2 Memory Usage

Besides the poor runtime performance, we already noticed in Section 3.1 that the transformed programs tended to consume significantly higher amounts of memory than equivalent pure Haskell programs. For this reason, some of the benchmarks had to be disabled since they exceeded the 32 GB of RAM available on our test machine and memory swapping started to dominate runtime performance. In this subsection, we are investigating the question of how the optimization affects the translated program's memory footprint.

Fortunately, Criterion can collect information about memory usage by hooking into APIs of Haskell's Garbage Collector (GC). To enable the collection of GC statistics, the `-T` option[8] of the Haskell Runtime System[9] (RTS) had to be enabled while running the benchmark. The GC statistics include amongst others the total number of bytes allocated during the run of the benchmark as well as the peak number of megabytes allocated since the start of the benchmark suite. While the peak allocation is of highest interest for us, this metric only produces meaningful results when each benchmark is run independently. Thus, we decided to configure Criterion to measure the total number of allocated bytes only and hoped that this metric correlates sufficiently with the memory utilization experienced by the user. The results for the largest problem instances are summarized in Table 5.2.

Table 5.2. Total number of bytes allocated by selected benchmarks with (*After*) and without (*Before*) worker/wrapper transformation plus improved sharing analysis enabled, respectively. The factor by which allocations have been reduced with respect to the unoptimized version is shown in the column labeled *Improvement*. For reference, the *Pure* column shows the number of allocated bytes for an equivalent unlifted program if applicable.

Benchmark	Before	After	Pure	Improvement
`AddNumInt10`	270 KB	171 KB	—	1.58 × less
`AddNumPeano10`	24 GB	195 MB	—	119 × less
`NaiveReverseIntList4096`	24 GB	808 MB	683 MB	28.9 × less
`NaiveReversePeanoList4096`	33 GB	11 GB	683 MB	3.28 × less
`PermSort12`	3 GB	3 GB	—	1.01 × less
`Primes`	233 MB	14 MB	—	16.8 × less
`QueensInt9`	6 GB	5 GB	36 MB	1.16 × less
`QueensPeano9`	31 GB	27 GB	55 MB	1.16 × less
`ReverseIntList4096`	11 GB	3 MB	361 KB	4570 × less
`ReversePeanoList4096`	21 GB	10 GB	295 KB	2.19 × less
`Select100`	607 MB	21 MB	—	29.0 × less
`TakInt27_16_8`	34 GB	359 MB	83 Bytes	93.3 × less
`TakPeano24_16_8`	94 GB	85 GB	25 MB	1.11 × less

[8] https://downloads.haskell.org/ghc/9.2.1/docs/html/users_guide/runtime_control.html#rts-options-to-produce-runtime-statistics
[9] https://gitlab.haskell.org/ghc/ghc/-/wikis/commentary/rts

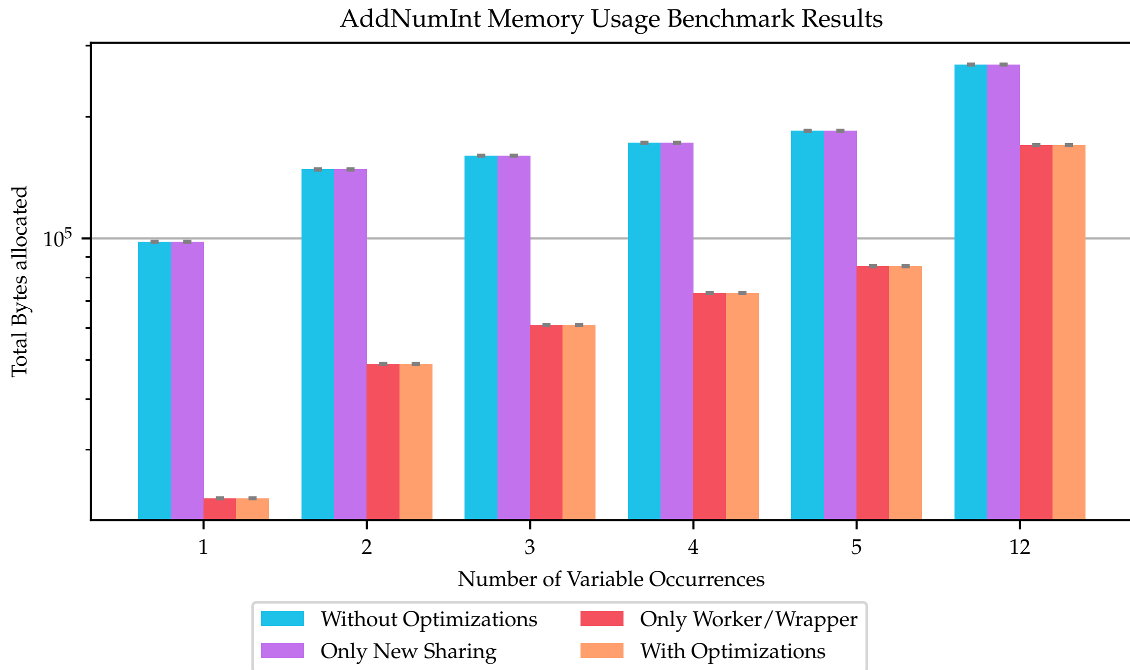


Figure 5.5. The memory usage of the `addSomeNum` benchmarks for integers.

If we compare Table 5.2 with Table 5.1 we can see that the improvement in memory usage is generally in the same order of magnitude as the runtime performance improvement of the same benchmark. Nevertheless, the absolute numbers are still much higher than the amount of memory equivalent pure Haskell programs would consume as becomes evident by comparison with the column labeled *Pure*. Next, we again discuss the results for the AddNum and Tak benchmarks in more detail

AddNum already consumed only a little amount of memory when integers were used to represent the numbers. As Figure 5.5 demonstrates, worker/wrapper transformation is responsible for a further reduction. The degenerate case saves 76 KB whereas the non-degenerate cases save 99 KB. Therefore, we conclude that the saved bytes originate from the memorization of `someNum`'s shared parameter `n`. After worker/wrapper transformation no `share` is needed because `someNum` is strict in `n`. The additional savings of the non-degenerate cases must stem from the elimination of monadic wrappers for the intermediate results of the additions.

On the other hand, Figure 5.6 shows that the memory usage of the benchmark's version using Peano numbers is enormous prior to optimization and grows exponentially. Similar to the runtime performance of the `AddNumPeano` benchmark, its memory usage is predominantly constrained by sharing. By using the new sharing analysis, the allocations return to linear growth whereas the worker/wrapper transformation has no effect in isolation. We suspect the same mechanism to be responsible for this phenomenon as for the similar behavior discovered for the runtime performance.

When both optimizations are enabled, the memory usage is reduced by two orders of magnitude. Still,

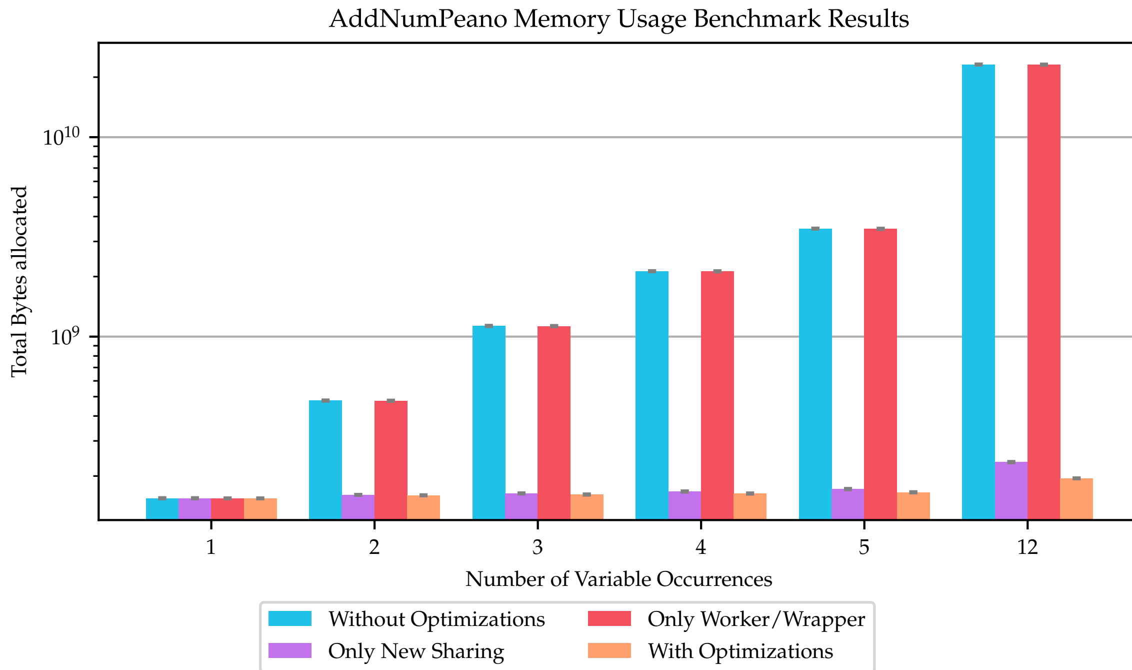


Figure 5.6. The memory usage of the `addSomeNum` benchmarks for Peano numbers.

the absolute values are in the range of megabytes and therefore a thousand times higher than in the integer variant of the benchmark. However, this is to be expected since Peano numbers are inherently a lot more expensive to represent than integers. Since all predecessors are stored as monadic values, the memory overhead is further increased.

Tak is a function that usually consumes very little memory. In fact, McCarthy (1979) explicitly points out the property, that the function can be made to run a long time without producing large numbers or using much stack. Figure 5.7 confirms this property for the integer variant of the benchmark when the function is not lifted. In this case, the memory usage is limited to a few bytes. While we do in fact expect the memory usage of the lifted version to be elevated, we are surprised by the extent of the increase and the residual memory overhead after optimization. All runs of the benchmark that involve the plugin exhibit a memory usage in the realm of gigabytes prior to worker/wrapper transformation and continue to allocate megabytes after being optimized. This behavior is particularly surprising as the memory allocations do not seem to have any negative impact on the runtime performance. For example, while the experimental usage of `unsafeBind` yielded the same runtime performance as the unlifted version of the benchmark, the lifted benchmark seems to consume a million times more memory. Until now, we have not found any convincing explanation for this phenomenon.

As we already established that the Peano representation of numbers is inherently more expensive than integers, it is not surprising to find the Peano variant of the benchmark to consume even more memory. Before lifting, we already observe a rapid growth of allocations in the range of megabytes. After applying the plugin, memory usage is increased to the range of gigabytes. The optimizations

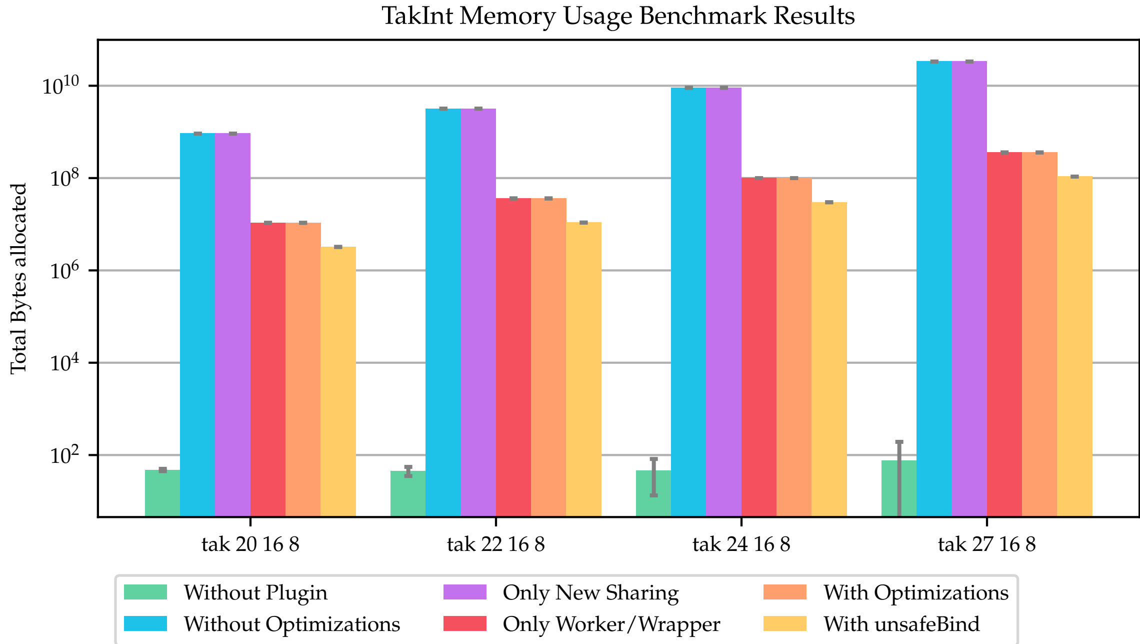


Figure 5.7. Memory usage of the `tak` benchmarks for integers.

can reduce the allocations by two-thirds but the absolute numbers remain in the gigabytes. In the most extreme cases, about 100 GB are allocated. As the measured metric represents the total number of bytes allocated, this amount is not necessarily allocated at the same time and memory might be freed in the meantime. However, the memory usage evidently exceeds the 32 GB of RAM installed in the test machine plus configured swap at some point during the execution of `tak 27 16 8` and causes a crash of the benchmark suite in effect.

5.3.3 Executable Size

Our optimization heavily relies on the use of `INLINE` pragmas. One caveat commonly associated with the excessive usage of inlining is an increase in executable size due to duplication of code by the compiler (Cooper and Torczon, 2012, pp. 461). To quantify the effect of our optimization on the size of generated executables, we measured the size of the compilation artifacts for the plugin's test and benchmark suites with and without worker/wrapper transformations enabled. Additionally, we repeated the measurement with dynamically linked versions of the executable to account for the overhead introduced by system libraries, the RTS and the plugin's dependencies.

The results of these measurements are presented in Table 5.3. There is one column for each compilation artifact and one row for the run with worker/wrapper transformation enabled and disabled, respectively. Additionally, the table provides the *Difference* in executable size between the two runs. This difference is expressed as a percentage relative to the measurement for the run with disabled worker/wrapper transformations in the final row.

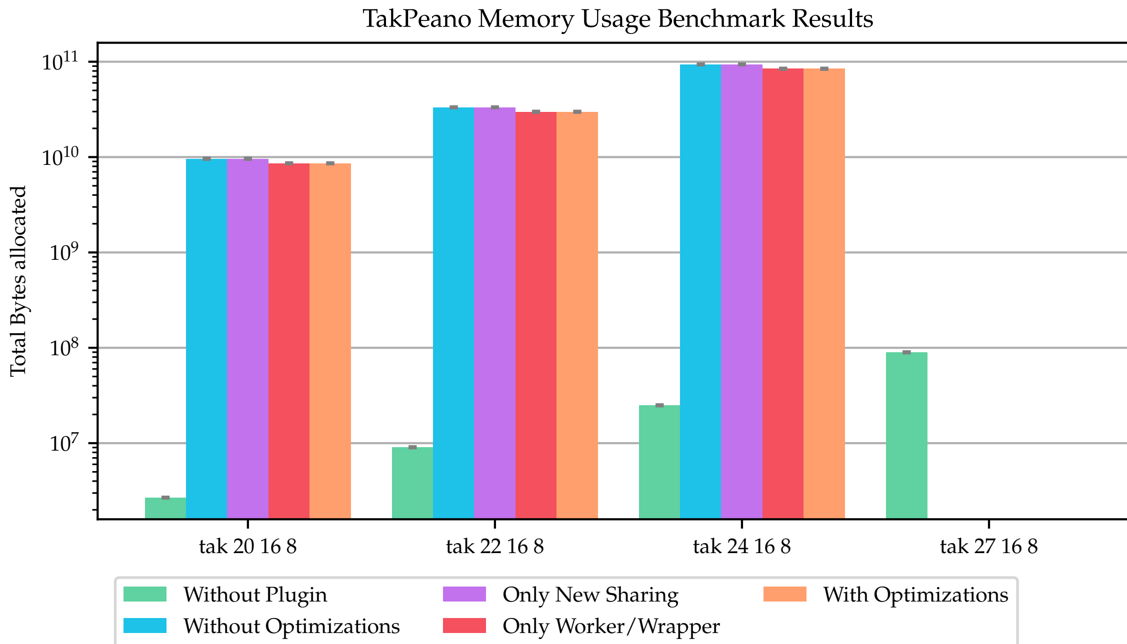


Figure 5.8. Memory usage of the `tak` benchmarks for Peano numbers.

As we can see, the worker/wrapper transformation exhibits only a minor influence on the size of generated executables. Contrary to our initial expectations, our observations even reveal a slight reduction in executable size for both the test and benchmark suites. This decrease can likely be attributed to optimizations of the GHC that are enabled through our worker/wrapper transformation. After inlining of a wrapper, the `return` operations for the lambda abstractions are duplicated. However, at the call site, there must be a corresponding bind operation unless the function is only partially applied. The GHC can now eliminate the `return` and bind due to our `RULE` pragma. The resulting code thus contains one bind operation less than the original code. The duplicated `return` only remains when the function is partially applied. Since most functions are usually fully applied, the overall number of bind operations is reduced and consequently the size of the executable.

Table 5.3. Executable sizes of the plugin's benchmark and test suites with and without worker/wrapper transformation for statically and dynamically linked executables.

	Staticly Linked		Dynamically Linked	
	Tests	Benchmarks	Tests	Benchmarks
Without Worker/Wrapper	26,472 KB	48,340 KB	667 KB	514 KB
With Worker/Wrapper	26,440 KB	48,319 KB	641 KB	502 KB
Difference	-32 KB	-21 KB	-25 KB	-12 KB
Difference %	-0.12 %	-0.04 %	-3.69 %	-2.27 %

Nevertheless, when put into perspective with the overhead introduced by system libraries, the run-time system and dependencies, the effect of this optimization is negligible. While the dynamically linked executables each reduced by between two and three percent in size, the difference in the statically linked executable's size amounts to less than half a percent.

Interestingly, the total number of bytes saved is largest for the statically linked executables, though. The additionally saved bytes likely result from the fact that the executables link against the plugin's library. The library is needed because the code generated by the plugin imports the plugin's monad and `Prelude` module. However, the `Prelude` is also worker/wrapper transformed and thus has a lower binary size after the transformation. Thus, the difference in size between the static executable includes the saved bytes from the `Prelude` whereas the dynamically linked executable only contains the saved bytes from the tests and benchmarks themselves. This theory suggests that there should be a constant difference in size between the static and dynamic executables. However, the static executable's size of the test suite decreased by 7.5 KB with respect to the corresponding dynamic library whereas the benchmark suite shows a 8.9 KB decrease of the same metric. Therefore, it should be noted, that the cause for the observed reduction in executable size cannot be conclusively explained by the worker/wrapper transformation alone unless dynamic linking inhibits inlining across package boundaries.

5.3.4 Compile Time

Up to this point, the benchmarks we have examined have predominantly indicated a positive influence of our optimizations on the respective metrics. However, to achieve these improvements, the plugin and the GHC have to perform additional work during the compilation process. We expect the following three aspects to entail longer compilation times:

1. the demand analysis,
2. the transformation of functions into workers and wrappers itself and
3. processing of inlined wrappers by the GHC.

To quantify the impact of these three aspects, we have measured the compile time of the test and benchmark suite. In contrast to Section 5.3.3, we did not record compilation times of the test and benchmark suite independently.

The results are presented in Table 5.4. The indicated times are the average amount of CPU time spent in user mode during the compilation process. The table also indicates the *Difference* in compile time between the run with and without worker/wrapper transformation and expresses this difference as a percentage relative to the measurement for the run with disabled worker/wrapper transformations in the last row. Since we compiled the tests and benchmarks at the same time, the table contains only one column for the cumulative compile time of both suites together.

As expected, the incorporation of worker/wrapper transformations led to an increase in compile time. Specifically, enabling worker/wrapper transformations during the compilation of the plugin's benchmark and test suites resulted in an increase of less than one second. In relative terms, this

Table 5.4. Average cumulative compilation time of the plugin's benchmark and test suites with and without worker/wrapper transformation.

	Average Compilation Time
Without Worker/Wrapper	43.7 s
With Worker/Wrapper	44.4 s
Difference	0.7 s
Difference %	1.66 %

corresponds to a one-and-a-half percent increase in comparison to the compilation time without worker/wrapper transformations.

While this increase in compilation time is only marginal, it's essential to consider that the plugin's benchmark and test suites are relatively small. For larger projects, the observed effects might be more pronounced. However, these effects are outweighed by the substantial performance improvements that the optimization provides at runtime.

5.4 Maintainability

In this final section of the evaluation, we will discuss which consequences the addition of the analyses and optimizations have on the future maintainability of the plugin. First, we consider how the improvements to the test and benchmark suite can help to prevent regressions. Then we discuss the challenges that arise from the newly added dependencies and the usage of the GHC API. Finally, we analyze the implementation's impact on the plugin's code size.

5.4.1 Tests and Benchmarks

Automated tests are important to ease the maintenance of a software project. Without automated tests, it is difficult to ensure that changes to the code do not break existing functionality. Our additions to the test suite therefore generally improve maintainability. Additionally, we configured a continuous integration pipeline based on GitHub Actions[10] that runs the test suite on every push to the plugin's GitHub repository[11] such that a broken test cannot go unnoticed in the future. However, the tests that have been automated can be considered a kind of integration test. While integration tests are important to gain confidence in the correct behavior of the tested system as a whole, unit tests are usually more effective in pinpointing the location of a bug and ensuring test coverage.

In principle, the benchmark suite can also be used to spot regressions that affect the implemented optimizations. However, unlike the test suite, the benchmark suite is not run as part of the continuous integration pipeline since it is too time-consuming. For the same reason, we do not expect the

[10] https://docs.github.com/en/actions
[11] https://github.com/just95/curry-ghc-language-plugin

benchmarks to be run regularly by maintainers.

5.4.2 Dependencies

The use of libraries such as `parsec`, `hspec` and `criterion` aided tremendously in the development of the plugin as well as its test and benchmark suite. However, each of the libraries comes with its own set of dependencies. These dependencies need to be maintained as well. Specifically, the versions of all transitive dependencies have to be listed as `extra-deps` in the `stack.yaml` file since the used Stack resolver `ghc-9.2.1` does not include them by default. If the GHC or any of the packages is updated to a newer version, we have to manually find a set of versions for all transitive dependencies that are mutually compatible.

5.4.3 GHC API

One of the reasons Prott (2020) states as a motivating factor for setting out to develop a Curry compiler by taking advantage of the GHC's Plugin system in the first place is the prospect of reducing the implementation effort. While the effort of implementing the plugin is indeed much lower than building a full-fledged compiler, the tight integration with the GHC poses additional challenges for the implementation and maintenance of the plugin.

The GHC's API is not stable and changes even between minor versions. During development, we regularly used Hoogle[12] to search for functions in the GHC's API that we can utilize. However, the index of Hoogle contains only the latest version of a package. Since the plugin is based on GHC 9.2.1 but the GHC is on version 9.6.3 at the time of writing, the search results often contain outdated information. For example, the demand signatures are stored in a data type **`StrictSig`** in GHC 9.2.1 but in a data type **`DmdSig`** in GHC 9.6.3. Besides the name of the data types, the structure of the underlying **`Demand`**s as well as their **`Outputable`** string representation has changed significantly. When upgrading the GHC version need to adapt the implementation to these changes inevitably will open the possibility for bugs to be introduced. As we have discovered in Section 5.1 an update of the GHC version has already caused problems in the past. We hope that the improvements to the test suite can help to mitigate some of these problems.

5.4.4 Code Size

A heuristic for the maintainability of a software project is the size of its code base. The more code a project encompasses, the more difficult it is to understand and maintain. Thus, we analyze in this section how many lines of code we have added over the course of the thesis.

When we started working on the thesis[13], the plugin's code base consisted of 10,100 lines of code across all `.hs` files including comments and blank lines. By the end of the thesis[14], the code base

[12] `https://hoogle.haskell.org/`
[13] Commit 4f0fd2e394bbd802e6d751062059865f599d2960
[14] Commit 2ec32618abe31a610ef3c17a957a7dce4ef52bd8

has grown to 18,656 lines of code. However, it is important to note, that large parts of the increase in code size can be attributed to the restructuring and formatting of the code that we performed in preparation of the implementation to get familiar with the code. In total, the addition of 3,069 lines of code can be attributed to the preparatory work[15]. Some of the steps that we took in this regard are listed below.

- ▷ We formatted the code using the Haskell code formatter Ormolu[16] and limited the line length to 80 characters where possible. The adoption of a code formatter helped during implementation by allowing us to focus on the actual functionality instead of having to manually shift code around to maintain readability.
- ▷ We split the code into several packages and more fine-grained modules. Previously the Curry GHC Language Plugin was a single package that was forked from the GHC Language Plugin and added the Curry plugin-specific code on top. By splitting the code into packages, we hope that it will be easier to port the optimizations to the other variants of the plugin. Splitting the modules makes the code easier to comprehend but since each module needs its own header, the code size increases.
- ▷ We streamlined the lifting process by adding a monad `TcState`. As the name suggests, this monad is similar to GHC's `TcM` monad but has state-like semantics whereas `TcM` has reader-like semantics. While this abstraction increased code size, the monad also helped to keep the code concise when we implemented optional passes for the optimization later on.

The implementation of the optimization itself primarily affected the general GHC Language Plugin's code. This part of the code increased from 9,965 lines to 12,745 lines. 2,298 lines of code have been added in the form of modules that exclusively deal with the analyses and worker/wrapper transformation. The remaining 482 lines resemble additions to the monadic lifting. The Curry-specific part of the code base is relatively small. It grew from 1,905 to 2,727 lines. The majority of the increase in code size can be attributed to the extension of the test suite and the addition of benchmarks. The prelude only increased by 71 lines of code. In summary, the total number of lines has almost doubled but the optimization code accounts for only 33% of the increase. Nevertheless, a much greater number of lines of code had to be touched to implement the optimization.

[15] Commit 1c862718d1c6592b8909f60672a9eaf13bc3203d

[16] `https://hackage.haskell.org/package/ormolu`

Chapter 6

Conclusion

In this final chapter, we conclude the thesis. In the first two sections, we summarize our work and discuss the limitations of the optimization approach and implementation thereof. Finally, we will be providing an outlook for future improvements to the plugin.

6.1 Summary and Results

The goal of this thesis was to improve the performance of the code generated by GHC Language Plugin for strict and pure functions. First, we familiarized ourselves with the Curry programming language which served as an example throughout the thesis. We then discussed the GHC's plugin system and how the GHC Language Plugin hooks into the compilation pipeline to monadically transform the source code. Next, we identified deep sharing and repeated monadic wrapping and unwrapping as the root causes of the performance problems exhibited by the generated code. We proposed the worker/wrapper transformation as a potential solution and convinced ourselves of the correctness of this optimization. In the main part of the thesis, we presented how we implemented the worker/wrapper transformation. To gather the strictness information required by this transformation, we were able to reuse the existing demand analysis of the GHC. Our evaluation in the final section showed that the worker/wrapper transformation is able to improve the performance of the generated code significantly. Our benchmarks suggest improvements in runtime, memory usage and even code size across a wide range of tasks while requiring moderate compile time costs. The degree of improvement depends on the depth of involved data structures, strictness of used functions and their effects. Moreover, we convinced ourselves of the correctness of our implementation by automatically testing the transformation on a large corpus of example programs.

In summary, we believe that we were able to achieve the primary objective of this thesis. Nevertheless, deep sharing prevails to pose fundamental problems for the generated code's performance. In the next section, we point out remaining limitations of our approach and shortcomings of our implementation before we close with an outlook on future work in the final section.

6.2 Limitations and Known Issues

While our evaluation demonstrates the potential of the worker/wrapper transformation to improve the performance of the generated code, our implementation has still issues that limit the performance gains in some situations.

As discussed in Section 4.1.4, our parser for demand signatures currently has to discard sub-demands due to restrictions of the GHC's API. As we have seen in Section 4.1.6, this can weaken inferred demands for functions using imported higher-order functions, for example. This problem is overshadowed by more fundamental restrictions of demand analysis that prevent effective analysis of higher-order functions without aid from the GHC's simplifier. Furthermore, we found in Section 4.1.5 that the demands of type class operations cannot be imported from lifted modules at all because the plugin discards the unfolding of dictionary functions.

While we have implemented experimental support for the insertion of `unsafeBind` as presented in Section 4.5.5 and demonstrated the effectiveness of this optimization to achieve runtime performance on par with an unlifted program's performance in Section 5.3.1, a big limitation of our implementation is that we cannot yet decide automatically whether it is safe to perform this optimization for a function or not. The effect analysis from Section 4.2 is insufficient in this regard and therefore currently not used by any part of the plugin.

To mitigate the problems associated with deep sharing, we presented an improved sharing analysis in Section 4.3. While our evaluation in Section 5.3.1 shows that the analysis is able to avoid unnecessary shares and therefore improves runtime performance, the effectiveness of the analysis is highly sensitive to the concrete syntactical representation of the program. For example, when the **`BangPatterns`** language extension is enabled, the user can mark strict arguments with an exclamation mark. The plugin generates a `seq` for such patterns. In consequence, the variable occurs one more time than usual on the right-hand side which means that a `share` maybe needs to be inserted that otherwise is not needed. While bang patterns are usually strategically inserted to improve performance, they can have the opposite effect when the plugin is used.

A final limitation that we need to point out is that the worker/wrapper transformation does not have any effect when the program is interpreted in GHCi. This is because we rely on GHC's simplifier to optimize the generated code. However, the simplifier is not run in GHCi. In consequence, it can be difficult to debug the plugin and example programs. We outline steps to workaround this issue in the plugin's documentation.

6.3 Future Work

To address the issues outlined in the previous section, more work is needed in the future. In this final section of the thesis, we point out possible extensions to our transformation and analyses as well as complementary work that has the potential to improve the performance of the generated code even further or simplifies the lifting process.

The most promising candidate to improve the performance of the Curry GHC Language Plugin is the integration of the memoized pull-tabbing-based monad by Hanus, Prott, and Teegen (2022). The monad is especially intriguing because it can prevent repeated deep sharing which continues to be the main performance bottleneck. Swapping the monad of the plugin should be quite easy. Unfortunately, the monad is Curry-specific. Other versions of the plugin therefore do not profit from this change.

To add support for parsing sub-demands, we either to need open a merge-request on the GHC's GitLab repository that adds the missing exports or bypass the restrictions of the API by making a

distinction between user-defined and generated demand signature annotations. The user-defined annotations would still not support sub-demands we could use the GHC's own serialization mechanism but for generated annotations.

Once support for sub-demands has been implemented, we might also want to investigate whether this additional information can be used for further optimizations. In Section 3.2.2 we have seen that the GHC uses sub-demands to unbox the fields of product data types. This technique could also be applied to our transformation but it is unclear how much of a benefit it would bring in our situation.

Another way to improve the results of demand analysis is to eta-expand the right-hand sides of function bindings. The demand analysis considers syntactic arguments of a function only. In consequence, we currently do not worker/wrapper transform point-free definitions. The GHC provides functions for the eta-expansion of Core expressions which we could use in the analysis phase. However, we would need to keep track of the number of lambdas abstractions added such that we can add the same number of lambda abstractions to the Haskell AST in order to prevent an arity mismatch.

In the previous section, we pointed out that sharing analysis is highly sensitive to the syntactical representation of the program. A more robust approach could be to reuse the cardinality intervals that are already part of the demand signatures. A `share` is not needed if the cardinality of an argument is `1`, **`S`** or `A`, because all of them guarantee that the argument is evaluated at most once. However, this means that we also need to record the demands that are placed on local variables.

There are also other ways in which we can reuse the machinery introduced for the implementation of the worker/wrapper transformation to optimize other parts of lifted programs. For example, **`BangPatterns`** in data constructors are currently implemented in terms of `seq`. The underlying fields still store a monadic value. Also regular data constructors have to be treated differently from regular functions because their result and partial applications are not monadic. We could either generate a wrapper function for data constructors or use **`AppSchema`**s directly to avoid special treatment of data constructors and the overhead of the residual monad in strict fields.

Finally, we want to suggest an improvement to the lifting process that has the potential to greatly simplify the code of the plugin itself. Currently, functions are replaced with their lifted version. The original definitions therefore do not exist after the lifting process. In consequence, the program is not type correct prior to lifting as it refers to imported functions that have been lifted and therefore a different type. The plugin deals with this situation by unlifting types in the constraint solver plugin. When we tried to incorporate the GHC's simplifier into our analyses we also ran into the problem that the unfoldings of imported functions are lifted. Furthermore, the need to restore the original demand signatures using an annotation is also a direct result of the fact, that imported functions are lifted and thus have a different demand signature. We, therefore, suggest to leave the original definitions in the program and rename the lifted functions instead. A similar approach is already taken for type-level declarations which are renamed by adding the **`ND`** suffix and leave the original declaration intact such that the program can be integrated with regular Haskell code. This also has the benefit that the unlifted functions can still be used and do not have to be implemented again if the same functionality is needed in the Haskell host application. The only problem is that there is no way to express the effects in unlifted code. Therefore, built-in operations such as `(?)` or `failed` have to be defined in terms of **`error`**. The plugin would not only profit due to the reduction of special cases but the original code can also be used to evaluate effect-free expressions with native performance.

Appendix A

Usage

To enable a GHC Language Plugin, its main module's name must be specified using the GHC's -fplugin flag. To configure the plugin, the GHC's -fplugin-opt flag can be used. The Curry GHC Language Plugin can be enabled and configured using the following pragmas for example. Available options are listed in Table A.1.

```
{-# OPTIONS_GHC -fplugin Plugin.CurryPlugin #-}
{-# OPTIONS_GHC -fplugin-opt Plugin.CurryPlugin:<option> #-}
```

Table A.1. Available plugin options.

Option	Description
`[no-]force-recompile`	Forces recompilation of the module even if neither the module nor its dependencies changed. This option is handy during development of the plugin because a module compiled with the plugin will not automatically recompile when the plugin is updated. Disabled by default except for tests and benchmarks.
`[no-]check-demand-signatures`	Enables/disabled the check whether demand signatures that have been calculated during demand analysis match the **ExpectedDemandSignature** annotations. Enabled by default.
`[no-]worker-wrapper`	Enables/disabled the worker/wrapper transformation. Enabled by default.
`[no-]effect-analysis`	Enables/disabled the effect analysis. Disabled by default.
`dump-original`	Dumps the original module before the plugin.
`dump-original-ev`	Dumps the original evidence bindings before the plugin.
`dump-original-inst-env`	Dumps the original class instance environment before the plugin.
`dump-original-type-env`	Dumps the original type environment before the plugin.
`dump-core`	Dumps the GHC core code before the plugin's analyses.
`dump-expanded-class-ops`	Dumps the GHC core code after expanding class operations.
`dump-deriving-errs`	Dumps errors that are reported during instance deriving.
`dump-pattern-matched`	Dumps the result of the pattern matching compiler.
`dump-worker-wrapper`	Dumps the result of the worker/wrapper transformation.
`dump-app-schemas`	Dumps the application schemas for worker functions.
`dump-inst-env`	Dumps the class instance environment after lifting.
`dump-demand-signatures`	Dumps the result of the demand analysis.
`dump-effects`	Dumps the result of the effect analysis.
`dump-final`	Dumps the final result after lifting the module.

The output of the `dump-demand-signatures`, `dump-app-schemas` and `dump-effects` options is formatted as follows.

```
Dump<Phase>
-----------
<function_name1> (<function_unique1>): <value1>;
<function_name2> (<function_unique2>): <value2>;
<function_name3> (<function_unique3>): <value3>;
...
```

The output is prefixed with the name of the function. Additionally, the function's name is followed by its **`Unique`** in parenthesis to disambiguate the entry. Sometimes the output contains multiple lines with the same function name and different uniques. This is the case because the output contains both the monomorphic and polymorphic identifiers. The value should be the same for both identifiers. Note that the output can also contain lines for locally defined functions. Therefore the printed names do not uniquely identify a binding.

Appendix B

Running Tests and Benchmarks

In this chapter, we provide instructions for how to run the tests and benchmarks that we used for the evaluation of our implementation in Chapter 5.

B.1 Running the Test Suite

To execute the automated tests presented in Section 5.1, run the following command in the root of the plugin's repository.

```
stack test :hspec
```

It is also possible to run only specific tests by passing the `--match` flag to Hspec. For example, the following command only runs the **`Coin`** example.

```
stack test :hspec --ta "--match Example.Coin"
```

Note that by default, the test suite recompiles all examples, even if only a single example is `-match`ed. This measure ensures that the examples were compiled with the latest version of the plugin. You can disable this behavior by unsetting the `-fplugin-opt=Plugin.CurryPlugin:force-recompile` option for the `hspec` test suite in `package/curry-ghc-language-plugin/package.yaml`.

B.2 Running the Benchmark Suite

To run the benchmarks presented in Section 5.3.1 and Section 5.3.2, run the following command in the root of the plugin's repository.

```
stack bench :criterion
```

Since the benchmark suite takes a long time to execute, we recommend running only those benchmarks that are of interest. The Criterion library accepts a pattern that is matched against the name of the benchmark to control which benchmarks to execute. For example, to run only the benchmarks for the manually translated and optimized versions of the **`TakInt`** example, run the following command.

```
stack bench :criterion --ba "TakInt/Manual"
```

Similar to the test suite, the benchmark suite recompiles all examples by default. You can disable this behavior by unsetting the `-fplugin-opt=Plugin.CurryPlugin:force-recompile` option for the `criterion` benchmark suite in `package/curry-ghc-language-plugin/package.yaml`.

By default, all benchmarks are executed with the worker/wrapper transformation enabled. To disable the woker/wrapper transformation, set the plugin's `no-worker-wrapper` option. For example, the following command runs all benchmarks without worker/wrapper transformation.

```
stack bench :criterion --ghc-options "-fplugin-opt=Plugin.CurryPlugin:no-worker-wrapper"
```

Due to the high peak memory consumption, we recommend to use at least 32 GB of memory, for the execution of the benchmarks without the worker/wrapper transformation enabled. We executed all benchmarks on a machine with an AMD Ryzen 7 3700X Prozessor with 32 GB of RAM running Ubuntu 22.04.3 LTS.

The results of the benchmarks are printed to the console and stored in `package/curry-ghc-language-plugin/benchmark.json`. The benchmark JSON report can be passed to the `thesis/tool/plot.py` script to extract the relevant data and generate plots of the benchmark results. To inspect the runtime performance results, Criterion also provides a convenient web interface that can be started by opening the generated `package/curry-ghc-language-plugin/benchmark.html` file in a browser.

To collect the data for our tables and plots, we used a script that runs the benchmark suite for one example at a time and each example once with worker/wrapper transformation enabled and disabled, respectively. By running the examples one by one, a failed run can be repeated more easily and the influence between examples is reduced. The script can be invoked as follows from the plugin repository's root directory.

```
./tool/bench.sh
```

B.3 Measuring Executable Size and Compile Time

To collect the data for Section 5.3.3 and Section 5.3.4 we created a Bash script that can be invoked as follows from the plugin repository's root directory.

```
./tool/bench-exe-size.sh
```

The script compiles the test and benchmark suite with and without worker/wrapper transformation enabled as well as with and without GHC's `-dynamic`[1] flag set. The compile time is measured using the `time`[2] command. To the prevent execution of tests and benchmarks from distorting the compile time measurement, the script sets the `--no-run-tests` and `--no-run-benchmarks` flags of the `stack`[3]

[1] https://downloads.haskell.org/ghc/9.2.1/docs/html/users_guide/phases.html#ghc-flag--dynamic
[2] https://man7.org/linux/man-pages/man1/time.1.html
[3] https://www.haskellstack.org/

build tool. Finally, the script locates the generated executables in `stack path --dist-dir` and prints their size using the `stat`[4] command and `%s` format string. In between the compilation runs, the script clears the build cache using the `stack clean -full` command.

For the measurement of executable sizes, a single run of the script is sufficient since the size of the executable is deterministic. For the measurement of compile-time, the script has been executed three times where each run yields two data points per compilation artifact.

The collected data can be found in `thesis/data/compile-time-and-exe-size.md`. The spreadsheet that has been used for calculations of averages, differences and percentages can be found in `thesis/data/compile-time-and-exe-size.ods`.

[4] https://man7.org/linux/man-pages/man1/stat.1.html

Appendix C

Peano Numbers

Throughout the thesis, we made extensive use of Peano numbers as an example data structure. Since we sometimes omitted the concrete implementation of common arithmetic operators for Peano numbers, we provide an overview of the operations on Peano numbers that we used in the first section. In the second section, we provide the previously omitted implementation of `tak` for Peano numbers.

First, let us recapitulate the definition of the data type that we are using to represent Peano numbers.

```
data Nat = O | S Nat
```

C.1 Common Operations

In the following listing, the definition of commonly used arithmetic operations on Peano numbers is given. In comments, we also provide mathematical justifications for each rule. Additionally, the strictness of each function is indicated as a demand signature.

```
{-# ANN add (ExpectedDemandSignature "<1L><ML>") #-}
add :: Nat -> Nat -> Nat
add O     n = n                      -- ∀n ∈ ℕ: 0 + n = n
add (S m) n = S (add m n)            -- ∀n,m ∈ ℕ: (m + 1) + n = m + n + 1

{-# ANN mult (ExpectedDemandSignature "<1L><L>") #-}
mult :: Nat -> Nat -> Nat
mult O     _ = O                     -- ∀n ∈ ℕ: 0 · n = 0
mult (S m) n = add n (mult m n)      -- ∀m,n ∈ ℕ: (m + 1) · n = m · n + n

{-# ANN leq (ExpectedDemandSignature "<1L><ML>") #-}
leq :: Nat -> Nat -> Bool
leq O _ = True                       -- ∀n ∈ ℕ: 0 ⩽ n
leq (S _) O = False                  -- ∀n ∈ ℕ: n + 1 > 0
leq (S m) (S n) = leq m n            -- ∀n ∈ ℕ: m ⩽ n ⇔ m + 1 ⩽ n + 1
```

C.2 Tak Implementation

In Section 5.3.1 we have given an implementation of `tak` for `Int` but omitted the corresponding Peano-based implementation due to their similarities. Below is the implementation of `tak` for Peano numbers.

```
{-# ANN takNat (ExpectedDemandSignature "<SL><L><SL>") #-}
takNat :: Nat -> Nat -> Nat -> Nat
takNat x y z =
  if x `leq` y
    then z
    else
      takNat (takNat (pred x) y z)
             (takNat (pred y) z x)
             (takNat (pred z) x y)
```

Appendix D

Worker/Wrapper Transformation using unsafeBind

In Section 3.3.5 we focused on the transformation scheme that our implementation applies by default if no experimental features are enabled. As discussed in Section 3.3.4, it is possible to improve upon the transformation by using an operation `unsafeBind`. The implementation of `unsafeBind` has been omitted so far. The implementation provided by the Curry GHC Language Plugin encapsulates the operation and throws an error if there is not exactly one result. Usually, the functions are not evaluated at all since the operator should be eliminated by the relevant GHC rewrite rules.

```
{-# INLINE [0] unsafeBind #-}
unsafeBind :: Nondet a -> a
unsafeBind mx = case runLazyT (unNondet mx) of
  Leaf x -> x
  _ -> error "unsafeBind: not a leaf"
```

While we also implemented experimental support for the generation of code that utilizes this operation, the guarantees provided by the effect analysis presented in Section 4.2 are not sufficient to decide whether the use of `unsafeBind` is safe. If the experimental translation scheme is activated, the generated worker and wrapper functions have the following shape where we use the same definitions as in Section 3.3.5.

```
f^ND :: Nondet (X_1^ND --> ... --> X_n^ND --> R^ND)
f^ND = return $ Func $ \x_1 ->
         ⋱
          return $ Func $ \x_n ->
            x_{s_1} >>= \y_{s_1} ->
              ⋱
                x_{s_m} >>= \y_{s_m} -> return (f^ND_worker y_1 y_2 ... y_n)

f^ND_worker :: Y_1^ND -> ... -> Y_n^ND -> R^ND
f^ND_worker y_1 ... y_n =
  let x_{s_1} = return y_{s_1}
      ⋮
      x_{s_m} = return y_{s_m}
    in unsafeBind rhs^ND
```

Bibliography

Antoy, Sergio and Michael Hanus (2005). "Declarative programming with function patterns". In: *International Workshop/Symposium on Logic-based Program Synthesis and Transformation*. DOI: 10.1007/11680093_2.

— (2006). "Overlapping rules and logic variables in functional logic programs". In: *Proceedings of the 22nd International Conference on Logic Programming*. ICLP'06. Seattle, WA: Springer-Verlag, pp. 87101. ISBN: 3540366350. DOI: 10.1007/11799573_9.

Braßel, B., M. Hanus, B. Peemöller, and F. Reck (2011). "Kics2: a new compiler from curry to haskell". In: *Proc. of the 20th International Workshop on Functional and (Constraint) Logic Programming (WFLP 2011)*. Springer LNCS 6816, pp. 1–18.

Brassel, Bernd and Sebastian Fischer (Sept. 2008). "From functional logic programs to purely functional programs preserving laziness". In: vol. 5836, pp. 25–42. ISBN: 978-3-642-24451-3. DOI: 10.1007/978-3-642-24452-0_2.

Cooper, Keith and Linda Torczon (2012). *Engineering a compiler: second edition*. ISBN: 978-0-12-088478-0.

Fischer, Sebastian, Oleg Kiselyov, and Chung-Chieh Shan (2011). "Purely functional lazy nondeterministic programming". In: *Journal of Functional Programming* 21.4-5, pp. 413465. DOI: 10.1017/S0956796811000189.

Gill, Andrew, John Launchbury, and Simon L. Peyton Jones (1993). "A short cut to deforestation". In: *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*. FPCA '93. Copenhagen, Denmark: Association for Computing Machinery, pp. 223232. ISBN: 089791595X. DOI: 10.1145/165180.165214.

Gill, Andy and Graham Hutton (Mar. 2009). "The worker/wrapper transformation". In: *Journal of Functional Programming* 19. DOI: 10.1017/S0956796809007175.

Goldberg, Adele and David Robson (1989). *Smalltalk-80: the language*. USA: Addison-Wesley Longman Publishing Co., Inc. ISBN: 0201136880. URL: http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf.

Hackett, Jennifer and Graham Hutton (2014). "Worker/wrapper/makes it/faster". In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ICFP '14. Gothenburg, Sweden: Association for Computing Machinery, pp. 95107. ISBN: 9781450328739. DOI: 10.1145/2628136.2628142.

Hanus, Michael (2012). "Improving Lazy Non-Deterministic Computations by Demand Analysis". In: *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*. Ed. by Agostino Dovier and Vítor Santos Costa. Vol. 17. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 130–143. ISBN: 978-3-939897-43-9. DOI: 10.4230/LIPIcs.ICLP.2012.130. URL: https://www.informatik.uni-kiel.de/~mh/papers/ICLP12.pdf.

Hanus, Michael, Kai-Oliver Prott, and Finn Teegen (2022). "A monadic implementation of functional logic programs". In: *Proceedings of the 24th International Symposium on Principles and Practice of Declarative Programming*. PPDP '22. Tbilisi, Georgia: Association for Computing Machinery. ISBN: 9781450397032. DOI: 10.1145/3551357.3551370.

Hanus, Michael and Fabian Skrlac (2014). "A modular and generic analysis server system for functional logic programs". In: *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation*

and Program Manipulation. PEPM '14. San Diego, California, USA: Association for Computing Machinery, pp. 181188. ISBN: 9781450326193. DOI: 10.1145/2543728.2543744.
Hanus, Michael and Finn Teegen (2020). "Memoized pull-tabbing for functional logic programming". In: *Workshop on Functional and Constraint Logic Programming*. DOI: 10.1007/978-3-030-75333-7_4.
Hanus (ed.), Michael (2016). *Curry: an integrated functional logic language (vers. 0.9.0)*. URL: https://www-ps.informatik.uni-kiel.de/currywiki/_media/documentation/report.pdf.
Hennessy, M. C.B. and E. A. Ashcroft (1977). "Parameter-passing mechanisms and nondeterminism". In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*. STOC 77. Boulder, Colorado, USA: Association for Computing Machinery, pp. 306311. ISBN: 9781450374095. DOI: 10.1145/800105.803420.
Hudak, Paul, John Hughes, Simon Peyton Jones, and Philip Wadler (2007). "A history of haskell: being lazy with class". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. HOPL III. San Diego, California: Association for Computing Machinery, pp. 1211255. ISBN: 9781595937667. URL: https://doi.org/10.1145/1238844.1238856.
Johnson-Davies, David (1986). "Six of the best against the clock". In: *Acorn User*. Retrieved September 3rd 2023, pp. 179, 181–182. URL: https://archive.org/details/AcornUser047-Jun86/page/n180/mode/1up.
Lämmel, Ralf and Simon Peyton Jones (2003). "Scrap your boilerplate: a practical design pattern for generic programming". In: *SIGPLAN Not.* 38.3, pp. 2637. ISSN: 0362-1340. DOI: 10.1145/640136.604179.
Lange, Max Joachim (2022). "Optimierung von generiertem monadischem code". Flensburg University of Applied Sciences.
Marlow, Simon and Simon Peyton-Jones (2012). *The glasgow haskell compiler*. Vol. 2. Lulu. URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2012/01/aos.pdf.
Marlow (ed.), Simon (Nov. 2010). *Haskell 2010 language report*. URL: https://www.haskell.org/definition/haskell2010.pdf.
McCarthy, John (1979). "An interesting lisp function". In: *Lisp Bull.* 3, pp. 68. DOI: 10.1145/1411829.1411833.
Najd, Shayan and Simon Peyton Jones (Oct. 2016). "Trees that grow". In: *Journal of Universal Computer Science* 23.
Peano, Giuseppe (1889). *Arithmetices principia: nova methodo*. Fratres Bocca. URL: http://archive.org/details/arithmeticespri00peangoog.
Peyton Jones, Simon (1987). *The implementation of functional programming*. Prentice Hall International. URL: https://www.microsoft.com/en-us/research/uploads/prod/1987/01/slpj-book-1987.pdf.
Peyton Jones, Simon and John Launchbury (Nov. 1998). "Unboxed values as first class citizens in a non-strict functional language". In: ISBN: 978-3-540-54396-1. DOI: 10.1007/3540543961_30.
Peyton Jones, Simon, Andrew Tolmach, and Tony Hoare (2001). "Playing by the rules: rewriting as a practical optimisation technique in ghc". In: *2001 Haskell Workshop*. ACM SIGPLAN. URL: https://www.microsoft.com/en-us/research/publication/playing-by-the-rules-rewriting-as-a-practical-optimisation-technique-in-ghc/.
Pickering, Matthew, Nicolas Wu, and Boldizsár Németh (2019). "Working with source plugins". In: *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell*. Haskell 2019. Berlin, Germany: Association for Computing Machinery, pp. 8597. ISBN: 9781450368131. DOI: 10.1145/3331545.3342599.
Prott, Kai-Oliver (2020). "Extending the glasgow haskell compiler for functional-logic programs with curry-plugin". MA thesis. Kiel University.
Sergey, Ilya, Simon Peyton Jones, and Dimitrios Vytiniotis (2017). "Theory and practice of demand analysis in haskell". In: URL: https://www.microsoft.com/en-us/research/wp-content/uploads/2017/03/demand-jfp-draft.pdf.

Sulzmann, Martin, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly (2007). "System f with type equality coercions". In: *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'07)*. ACM, pp. 53–66. ISBN: 1-59593-393-X.

Teegen, Finn, Kai-Oliver Prott, and Niels Bunkenburg (2021). "Haskell^{-1}: automatic function inversion in haskell". In: *Proceedings of the 14th ACM SIGPLAN International Symposium on Haskell*. Haskell 2021. Virtual, Republic of Korea: Association for Computing Machinery, pp. 4155. ISBN: 9781450386159. DOI: 10.1145/3471874.3472982.

Wadler, P. and S. Blott (1989). "How to make ad-hoc polymorphism less ad hoc". English. In: *POPL '89 Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, pp. 60–76. ISBN: 0-89791-294-2. DOI: 10.1145/75277.75283.

Wadler, Philip (1987). "Efficient compilation of pattern-matching". In: *The Implementation of Functional Programming Languages*. Ed. by Simon Peyton Jones. Prentice-Hall, pp. 78–103. DOI: 10.1016/0141-9331(87)90510-2.

— (1990). "Comprehending monads". In: *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*. LFP 90. Nice, France: Association for Computing Machinery, pp. 6178. ISBN: 089791368X. DOI: 10.1145/91556.91592.

Wieczerkowski, Fredrik (2021). "A compiler for curry based on a ghc plugin". Kiel University.