The Worker-Wrapper Transformation

Getting it right and making it better

Jennifer Hackett

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

May 2016

Abstract

A program optimisation must have two key properties: it must preserve the meaning of programs (correctness) while also making them more efficient (improvement). An optimisation's correctness can often be rigorously proven using formal mathematical methods, but improvement is generally considered harder to prove formally and is thus typically demonstrated with empirical techniques such as benchmarking. The result is a conspicuous "reasoning gap" between correctness and efficiency. In this thesis, we focus on a general-purpose optimisation: the worker/wrapper transformation. We develop a range of theories for establishing correctness and improvement properties of this transformation that all share a common structure. Our development culminates in a single theory that can be used to reason about both correctness and efficiency in a unified manner, thereby bridging the reasoning gap.

Acknowledgements

Many people have played a part in the development of this thesis, and many more have played a part in the development of myself as a person. My personal and professional life for the past five years has not been without hardship, and I would not be here if not for the support of friends and colleagues. This is by no means an exhaustive list, but I would like to mention a few people in particular.

Professionally, I would like to thank my supervisor Graham Hutton: for his advice, for his assistance and (perhaps most crucially) for having me as a student in the first place. I would also like to thank my current and former colleagues at the Functional Programming Laboratory in Nottingham who have made it such an interesting and enjoyable place to work. Particular thanks are owed to my PhD "siblings" Laurence Day and Liyang Hu, who have been consistently supportive and understanding.

Personally, I would like to thank my friends, both in Nottingham and the wider world, for their support, inspiration, humour and tolerance for excruciating puns. Most of all, I would like to thank my partner Hazel Smith, whose support, care and love throughout the time we have been together has been invaluable.

Contents

Part I: Background

Working it out

Chapter 1

Introduction

"Begin at the beginning and go on till you come to the end: then stop."

— the King, Alice's Adventures in Wonderland

In programming, as in many things, there is often a tension between clarity and efficiency. Programs that are written with clarity in mind often use inefficient abstractions to make things conceptually easier, while efficiency often requires writing "close to the bare metal", i.e. avoiding abstractions in favour of efficient low-level operations. When abstraction is less of an issue, there are still tricks and twists to writing efficient code that are often incomprehensible to a casual reader. Put simply, programs that are written for people to read are often very different from programs that are written for machines to run.

To bridge the gap between clarity and efficiency, we need optimisations. Optimisations are processes, usually applied during the compilation process, that transform inefficient programs into more efficient ones. This allows a programmer to write programs in a style that is less efficient but more clear, secure in the knowledge that the inefficiencies will be dealt with by the compiler.

Unfortunately, this picture is far too good to be true. Many complex program optimisations cannot be applied automatically, instead requiring human insight to see

when they are applicable. Even among those that can be automated an efficiency gain is often not guaranteed, having been only verified through empirical techniques such as benchmarking. Clearly, more research is required before we can live in the utopian world we have described.

This thesis goes some way toward solving this problem in the context of one particular program optimisation: the *worker/wrapper transformation*. This transformation is a very general pattern that captures a wide range of optimisations. The generality of the worker/wrapper transformation means not only that it is widely applicable, but that it lends itself to general theoretical techniques that will likely be applicable elsewhere. Hence, research on the worker/wrapper transformation can likely lead to theoretical advances in other areas of program optimisation. In particular, this thesis demonstrates the practicality and generality of several techniques that have previously been neglected in the field of program optimisation.

The worker/wrapper transformation already comes with a significant amount of theoretical work behind it tackling the problem of correctness. This thesis extends this correctness work, as well as making a major contribution toward the *improvement* side of the worker/wrapper transformation.

1.1 Dualities

This thesis touches on a number of interesting *dualities*: situations where two things are somehow simultaneously opposed and connected. The idea of duality can be taken philosophically as a broad notion of two-ness that appears in many places: for example, mind versus matter (Descartes, 1641), life versus death (Anon., C. 3500–1900 BCE), particles versus waves (de Broglie, 1924). More specifically, there are *mathematical* dualities, which arise when one mathematical system is shown to correspond directly to the "inverse" of another.

Mathematical dualities arise everywhere. For example, boolean algebra has the

duality of the De Morgan laws, where conjunction and disjunction seemingly play opposite roles but can be replaced by one another by applying negations. As such, boolean algebra can be seen as its *own* dual.

Dualities are important for several reasons. Firstly, observing a duality between two systems or ideas can lead to important observations about the systems or ideas themselves, as properties of one give us properties of the other. Secondly, mathematical duality provides a handy proof technique, where in order to prove something about one particular system one can prove the dual statement about the dual system.

The first duality in this thesis is *categorical* duality. Categorical duality is a phenomenon that arises from the generality of category theory, where every category gives rise to a *dual* category where the direction of arrows is reversed. This allows properties of categories to be dualised as well, and provides the useful result that if a given property holds of all categories, so does the dual property. This thesis exploits duality in its treatment of categorical folds and unfolds.

The second duality in this thesis is the duality between *correctness* and *improvement*. These are the two sides of any successful program optimisation: an optimised program must be correct, and it must be somehow "better". In this thesis, we observe that while these two aspects are seemingly disparate, they are in fact closely linked.

The third duality in this thesis is the duality between *operational* and *denotational* semantics. Operational semantics is the idea of defining the meaning of a program by what it *does*, i.e. by its behaviour. Denotational semantics, however, is the idea of defining the meaning of a program by what it *is* in some mathematical model. This is related to the first duality as operational semantics are often put in terms of *coalgebras*, and denotational semantics in terms of *algebras*, these concepts being categorically dual. This is also related to the second duality, as denotational approaches frequently suit themselves well to reasoning about correctness while operational approaches are more suited to reasoning about improvement. However, this is by no means a hard-and-fast rule. This thesis uses both operational and denotational ap-

proaches to reasoning about programs, as well as noting key similarities and differences between the approaches.

The fourth duality in this thesis is the duality between clarity and efficiency, as mentioned earlier in this chapter. This duality reflects the twin purposes of source code, which is meant to be read by both humans and computers. This is linked to the duality between denotational and operational semantics, as declarative code that lends itself to denotational reasoning is typically more clear, while imperative code that lends itself to operational reasoning is typically more efficient.

Finally, this thesis can itself be viewed in two opposite ways. On the one hand, it can be viewed as a thesis primarily about the worker/wrapper transformation that explores this transformation through the lens of various different theories and techniques. On the other hand, it can be viewed as being an exploration of those theories and techniques as viewed through the lens of the worker/wrapper transformation. In other words, the thesis has its *own* duality.

It should be noted that while duality is a powerful concept, it is rarely the whole story. Even in cases of strict mathematical duality, it is often valuable to examine the differing *pragmatic* implications of both sides. Things are often dual, but they are rarely "just" dual.

1.2 The Organisation of This Thesis

In keeping with the theme of duality, the bulk of this thesis is divided into two parts. To be specific:

- Part I consists of two chapters. The first chapter is this introduction, while the second chapter introduces and demonstrates the worker/wrapper transformation by way of examples. We demonstrate the worker/wrapper transformations's original form of *unboxing* and give a history of the transformation.

- Part II presents the first half of the work, addressing the problem of proving the correctness of the worker/wrapper transformation. This consists of three separate correctness theories: one for programs written using general recursion, one for programs written in the form of an unfold and a generalised theory based on the categorical notion of *strong dinatural transformations*.

- Part III presents the second half of the work, addressing the problem of reasoning about program *improvement*. In other words, this part is concerned with proving that the worker/wrapper transformation improves performance. We present two improvement theories: an operational theory for programs written using recursive let bindings and a theory that models improvement using *preorder enriched categories*.

- Part IV serves as a conclusion, in which we summarise the contributions of the thesis, compare and contrast the various worker/wrapper theories we presented and discuss possible avenues for future work.

Each of the two central parts consists of an initial chapter introducing the problem addressed in that part, chapters discussing the work of this thesis, and finally a concluding chapter.

1.3 Contributions

We make the following contributions:

- In Chapter 5 we give a novel presentation of the worker/wrapper transformation for programs written in the form of an unfold, and discuss the ways in which it compares to its dual theory for folds and the existing theory for least fixed points.

- In Chapter 6 we present a general worker/wrapper transformation based on the category-theoretic notion of strong dinaturality, along with a correctness proof.

This establishes strong dinaturality, a generalisation of fusion, as the core of the worker/wrapper transformation.

- In Chapter 8 we introduce the technique of using preorder-enriched categories to reason about notions of program improvement, taking inspiration from earlier work on similar approaches to program *refinement*.
- In Chapter 9 we use improvement theory (Moran and Sands, 1999a) to verify that, under certain natural conditions, the worker/wrapper transformation for least fixed points does not make programs slower. This not only proves an important property of the worker/wrapper transformation, it also demonstrates the application of a theory that has been underappreciated in the literature.
- In Chapter 10 we extend the presentation of the worker/wrapper transformation based on strong dinaturality to one based on *bilax* strong dinaturality, resulting in a generalised theory giving conditions under which the generalised worker/wrapper transformation does not make programs worse.

Some of this work has been published in earlier papers. Specifically:

- The material in Chapter 5 on the worker/wrapper transformation for unfolds is based on work published in the post-proceedings of the ACM Symposium on Implementation and Application of Functional Languages (Hackett, Hutton, and Jaskelioff, 2013).
- The material in Chapter 9 on the worker/wrapper theorem for call-by-need improvement is based on work presented at the ACM SIGPLAN International Conference on Functional Programming (Hackett and Hutton, 2014).
- The material in Chapters 6 and 10 on worker/wrapper for (bilax) strong dinatural transformations is based on work presented at the ACM/IEEE Symposium on Logic In Computer Science (Hackett and Hutton, 2015).

The author of this thesis was the lead author of all of these papers.

1.4 Prerequisites

Much of this thesis requires basic knowledge of category theory, up to natural transformations. Any introductory text will go this far, but the standard text is Categories for the Working Mathematician by Mac Lane (1971). We assume knowledge of program semantics including the basics of domain theory and call-by-need evaluation; for an introduction to these concepts, see Reynolds (1998b). Program examples are written in Haskell syntax, so basic knowledge of Haskell will also be useful. There are numerous introductory texts on Haskell, including Hutton (2016) and Bird (1998).

Chapter 2

Worker/Wrapper By Example

> *"In every job that must be done, there is an element of fun. You find the fun, and: snap! The job's a game!"*
>
> *— Mary Poppins, A Spoonful of Sugar*

This thesis is primarily about the worker/wrapper transformation. Broadly speaking, the worker/wrapper transformation is a program *factorisation* where an inefficient program is split into two parts. The first part, the "worker", does the bulk of the work, hopefully in a more efficient way and generally at a new type. The second part, the "wrapper", is an adaptor that allows the worker to be used in the same context as the original program. This general idea is illustrated in Figure 2.1.

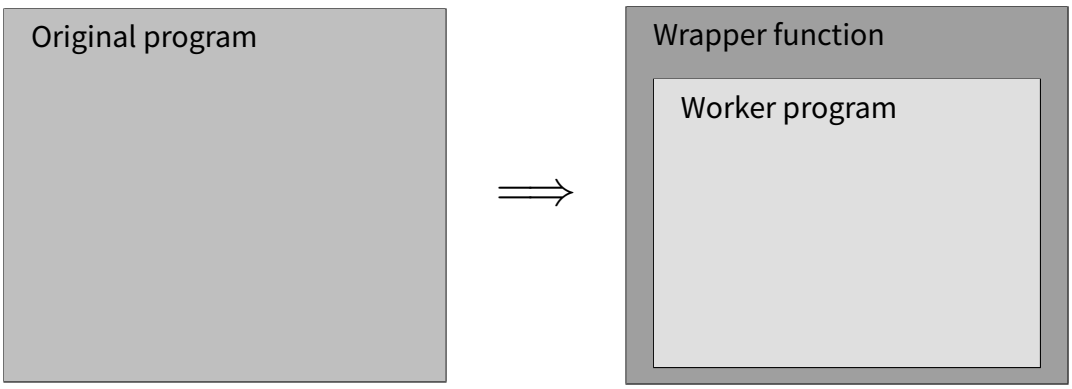


Figure 2.1: The Worker/Wrapper Transformation in Pictures

2.1 A Change of Representation

The general idea behind the worker/wrapper transformation is that costs can be reduced by working with a new representation of data that supports more efficient operations. Letting A be a type corresponding to the original representation and B be a type corresponding to the new representation, this representation change is captured by a pair of functions *abs* and *rep*, where *rep represents* something of type A in type B and *abs abstracts* from type B to type A.

We can demonstrate this idea with an example. We begin with the following implementation of the *reverse* function, which reverses a list:

$$
\begin{aligned}
&\mathit{reverse} :: [a] \to [a] \\
&\mathit{reverse}\ [\,] \qquad = [\,] \\
&\mathit{reverse}\ (x : xs) = \mathit{reverse}\ xs +\!\!+ [x]
\end{aligned}
$$

This can be improved by applying the worker/wrapper transformation, factorising into a form that uses a secondary function *revcat*:

$$
\begin{aligned}
&\mathit{reverse}\ xs = \mathit{revcat}\ xs\ [\,] \\
&\quad \mathbf{where}\ \mathit{revcat} :: [a] \to [a] \to [a] \\
&\qquad \mathit{revcat}\ [\,]\ ys \qquad = ys \\
&\qquad \mathit{revcat}\ (x : xs)\ ys = \mathit{revcat}\ xs\ (x : ys)
\end{aligned}
$$

The new function *revcat* is the worker. It performs the same task as the original reverse — that of reversing elements — but uses an accumulating parameter to do this in a more efficient way. Equivalently, this can be viewed as producing the reversed output in a form known as a *difference list* (Hughes, 1986), where a list is represented by the function that prepends its elements to another list. In either case, the function *revcat* is *wrapped* by an application to the empty list, resulting in a function that is equivalent to the original *reverse* function, but more efficient. (To be precise, the original function takes quadratic time, while the factorised version is linear.)

The worker/wrapper transformation has many presentations, each with their own set of preconditions for correctness. However, common to all of these presentations is the idea of factorising a program into two parts to facilitate a change of type. The type conversions this requires carry a cost, but the hope is that the costs saved by using the new type will be more than enough to offset this. This idea is at least informally understood by a large number of programmers, though the formalised forms of the transformation are less widely-known.

2.2 Unboxing

The name "worker/wrapper" comes from when the transformation was originally presented in the limited form of *unboxing* by Peyton Jones and Launchbury (1991). In strict languages, values such as integers can be stored in memory using their standard representations. In lazy languages like Haskell, however, an integer stored in memory could either be an actual value or a stored *computation* that has yet to be fully evaluated. Most implementations use some form of tagging to indicate which is the case. These tagged representations are called *boxed* values, while the raw representations as in strict languages are called *unboxed*. In Haskell, unboxed types, functions and values are typically denoted with a hash (#) sign.

Unboxing is a transformation of Haskell programs that replaces the usual boxed primitive types with "unboxed" types that are forced to be treated strictly. Essentially we replace the standard boxed type Int with its unboxed counterpart $Int\#$, which can only be used when fully-evaluated. We can use these types as though we had the following data declaration for Int:

$$\mathbf{data}\ Int = I\#\ Int\#$$

Unboxing can therefore be used to avoid undesired side-effects of laziness. Consider this implementation of the factorial function that uses an accumulating parameter:

$$
\begin{aligned}
&\mathit{afac} :: \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
&\mathit{afac}\ n\ r = \mathbf{case}\ n\ \mathbf{of} \\
&\quad 0 \to r \\
&\quad \_ \to \mathit{afac}\ (n-1)\ (n*r)
\end{aligned}
$$

Because nothing in *afac* forces the evaluation of the parameter r, large products build up during evaluation, as demonstrated by the following evaluation of $\mathit{afac}\ 3\ 1$:

$$
\begin{aligned}
&\quad \mathit{afac}\ 3\ 1 \\
&\to \mathit{afac}\ (3-1)\ (3*1) \\
&\to \mathit{afac}\ 2\ (3*1) \\
&\to \mathit{afac}\ (2-1)\ (2*(3*1)) \\
&\to \mathit{afac}\ 1\ (2*(3*1)) \\
&\to \mathit{afac}\ (1-1)\ (1*(2*(3*1))) \\
&\to \mathit{afac}\ 0\ (1*(2*(3*1))) \\
&\to (1*(2*(3*1))) \\
&\to (1*(2*3)) \\
&\to (1*6) \\
&\to 6
\end{aligned}
$$

Using unboxing, this can be factorised to the following:

$$
\begin{aligned}
&\mathit{afac}\ n\ r = \mathbf{case}\ n\ \mathbf{of}\ \mathit{I\#}\ n\# \to \mathbf{case}\ r\ \mathbf{of}\ \mathit{I\#}\ r\# \to \mathit{I\#}\ (\mathit{afac\#}\ n\#\ r\#) \\
&\quad \mathbf{where}\ \mathit{afac\#}\ n\#\ r\# = \\
&\qquad \mathbf{case}\ n\#\ \mathbf{of} \\
&\qquad\quad 0\# \to r\# \\
&\qquad\quad \_ \to \mathbf{case}\ (n\# -\# 1\#)\ \mathbf{of} \\
&\qquad\qquad\qquad \mathit{n1\#} \to \mathbf{case}\ (n\# *\# r\#)\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\qquad \mathit{r1\#} \to \mathit{afac}\ \mathit{n1\#}\ \mathit{r1\#}
\end{aligned}
$$

The subprogram *afac#* has parameters and result of type $\mathit{Int\#}$, rather than the standard Int. In particular, this means that the parameter $r\#$ must be treated strictly, so

the large products that plagued the first implementation cannot be built. Note that the only unboxed values that appear in expressions are those that have already been evaluated by a case statement, ensuring that no unboxed thunks ever appear.

2.3 History of the Worker/Wrapper Transformation

The full version of the worker/wrapper transformation was first presented by Gill and Hutton (2009). This presentation was based on least fixed points, and provided a definition of the worker program that required some simplification in order to reap the full benefits of the transformation. The authors presented a "worker/wrapper fusion theorem" to assist in this process.

The next stage in the history of the worker/wrapper transformation was the presentation due to Hutton, Jaskelioff, and Gill (2010). This presentation was based on the category-theoretic notion of folds and had a four-pointed lattice of related preconditions, all but one of which was strong enough to imply the correctness of the transformation. This gave the treatment a degree of theoretical elegance, but unfortunately meant that it was incompatible with the previous presentation.

Once these two separate theories of the transformation had been presented, the question was raised as to whether they could be unified. A partial answer was given by Sculthorpe and Hutton (2014), who gave a uniform *presentation* of the worker/wrapper transformation for both least fixed points and folds. Furthermore, while they could not unify the two theories entirely, they did show some compelling links between the two.

Notwithstanding the implementation of unboxing in GHC, the question remained as to whether the general form of the transformation could be automated. An interactive program transformation tool called HERMIT was developed by Farmer, Gill, Komp, and Sculthorpe (2012); this tool is capable of applying the worker/wrapper transformation, among others (Sculthorpe, Farmer, and Gill, 2013). However, there

is not yet an implementation of the general worker-wrapper transformation that can be applied fully automatically.

2.4 Worker/Wrapper Prehistory

The general form of the worker/wrapper technique appeared long before it was known by that name. Possibly the earliest example of a worker/wrapper-like technique is in Milner's work on *program simulation* (Milner, 1971). This work deals with programs in the form of state machines; a *weak simulation* between state machines A and B is a relation R between the states of A and the states of B such that if $(x, y) \in R$ and $x \to x', y \to y'$ then $(x', y') \in R$. These machines have privileged input and output states that the simulation relation is required to respect, in the sense that input states are related only to input states and the same for outputs.

If it is also the case that the relation R's action on input states can be represented by a function $f : \mathit{inputs}\ A \to \mathit{inputs}\ B$ and its action on outputs by a function $g : \mathit{outputs}\ B \to \mathit{outputs}\ A$, it is a *strong simulation*. In this case, the program represented by A can be factorised into $A' = g \circ B' \circ f$, where A' and B' are the partial functions represented by the state machines A and B. In other words, this theory gives us conditions under which a program A can be factorised into a worker B and functions f, g that effect a change of data representation: a worker/wrapper transformation. However, unlike the worker/wrapper transformations we discuss in this thesis, Milner's work is not primarily concerned with *deriving* the factorised program, only proving it correct.

Related to this is Hoare's work on *data representations* (Hoare, 1972). This work is based on the idea of *abstract data*, where the representation of data is postponed until after after the algorithm has been designed. The data is treated as an abstract mathematical object with a set of operations, and correctness of the concrete program is proved by verifying the correctness of both the abstract algorithm and the

implementation of the data representation. In order to verify the correctness of the implementation, we must have a function that maps concrete data to its abstract meaning, akin to our function abs. However, there is no explicit definition of rep; instead, the abs function is used to create logical pre- and postconditions that can then be verified via Hoare logic. Like Milner's work, this work is not primarily concerned with the derivation of the concrete program, although it does go further in that the work of creating the concrete program is straightforward once the concrete implementation of the abstract operations has been produced. This work is generally applied to imperative languages, in contrast with the worker/wrapper transformation's functional style.

One particular worker/wrapper-style technique is popularly known in the programming language community by another name: "defunctionalisation" (Reynolds, 1998a). Defunctionalisation is the process by which one transforms a program that uses higher-order functions to one that is based entirely on first-order functions. This is useful when building interpreters, as an initial implementation that uses functions to represent constructs in the interpreted language can be transformed into one that uses first-order data for the same purposes. This process generally involves creating a function to "apply" these function objects, i.e. a function that converts the concrete implementation of a function to its abstract representation. Once again, this is just like the abs function we use here. However, this generally results in a type that is "smaller", in the sense that there are some functions that could be represented by the original function type that are no longer representable, simply because those extra possible values were unneeded. This differs from the case of worker/wrapper here, where the concrete type B is generally *larger* than the abstract type A.

Another interesting proto-worker/wrapper application is in work on the logic programming language Prolog by de Bruin and de Vink (1990). The authors create two semantics of Prolog, one operational and one denotational, and prove them equivalent. The point of interest here is that they define a notion of *retract* between CPPOs

where D' is a retract of D if there are functions $i : D \to D'$ and $j : D' \to D$ such that $j \circ i = id$. If we replace i for rep and j for abs we can see that this is exactly the same as the precondition of the worker/wrapper transformation. The authors go further, however, developing conditions under which $fix\ f = j\ (fix\ g)$, where fix is the least fixed point operator. These conditions are not the same as those of Gill and Hutton (2009), but the idea is clearly the same: $fix\ f$ has been factorised into worker $fix\ g$ and wrapper j. Note that this is not the same as simple fixed point fusion for precisely the same reason that worker/wrapper for fixed points differ from simple fusion: the requirement that the data representation be *faithful*.

Part II: Correctness

Getting it right

Chapter 3

Introduction

"Measure twice, cut once."

— English proverb

This part of the thesis is concerned with the problem of proving the *correctness* of various forms of the worker/wrapper transformation. As stated before, this is simply the requirement that the transformation should not change the meaning of a program. Clearly this is an important property for any optimisation. However, a correctness theory can be used in one of two ways. Obviously, given two programs we can ask if one is a correct transformation of the other. But perhaps less obviously, given one program, we can obtain conditions that we can use to *derive* the correctly transformed program.

Before we can begin to ask the question of correctness, we must first decide what the programs mean to begin with; this is a problem for the field of *program semantics*. The bulk of this chapter is therefore given to an overview of this field. After this introductory material, the main body of this part of the thesis consists of three chapters. Each chapter presents the correctness theory for one form of the worker/wrapper transformation. The first such theory is for the worker/wrapper transformation for least fixed points, a denotational theory that can be used to reason about generally-recursive functions. The second theory is for the worker/wrapper transformation for

unfolds, a pattern of structured recursion (actually, *co*recursion) where a single seed is used to generate a potentially-infinite data structure. The final theory is a worker/wrapper theory for *dinatural transformations*, a categorical concept that aims to unify the prior worker/wrapper theories. After this, there is a short concluding chapter.

3.1 Background

In this section, we review the background of the study of program semantics, focusing on the three areas of *denotational*, *operational*, and *categorical* semantics. Each of these areas comes with its own set of notions of what a program can mean and its own techniques for proving program correctness.

3.1.1 Denotational Semantics

The key idea behind denotational semantics is that a program's meaning is what the program *is*. A denotational semantics is defined firstly by choosing a set of mathematical values, called the *domain*, which will contain all the possible meanings of programs. Programs are then *interpreted* by means of some mapping from the syntax to this domain, so that every program is assigned a single value in the domain. This mapping will typically be defined recursively on the structure of the program, so that the meaning of a program is some function of the meaning of its subprograms. We say that a semantics defined in this way is *compositional*.

Under denotational semantics, two programs are considered equivalent if they map to the same value in the domain. It is therefore natural to prove correctness of a transformation by proving the equality of these values. Thus, denotational semantics lends itself to techniques of *equational reasoning*, where two program terms are shown to be equivalent by presenting a sequence of terms with equalities between adjacent terms. Denotational approaches also lend themselves well to *calculational* programming, where algebraic techniques are used to derive a program that satisfies

some specification. Both techniques are used extensively in this thesis.

The concept of denotational semantics was first developed by Scott and Strachey (1971), who proposed interpreting an imperative program as a function taking an initial state as input and producing a final state as output. Furthermore, their approach could deal with recursively-defined programs by making the set of possible states into a *lattice*, a kind of ordered set where every pair of elements has both a least upper bound and greatest lower bound. This was based on an idea from previous work by Scott (1970), where data types were interpreted as partially-ordered sets.

The types-as-lattices approach was subsequently refined to semantics based on *complete (pointed) partial orders* (CPPOs) (Scott, 1982), which are partial orders with a different completeness property that implies that functions have least fixed points. This structure also provides an algorithm to calculate such fixed points, in the form of the *Kleene fixed point theorem* (Kleene, 1952). CPPO models form the basis of most modern denotational semantics for functional languages. An important exception to this is total languages, which do not allow unrestricted use of recursion.

3.1.2 Operational Semantics

In contrast to denotational semantics, operational semantics defines a program not by what it is, but by what it *does*. In this approach, the meaning of a program is the sequence of steps, inputs and outputs that must be performed to calculate its result. These steps may or may not be the steps that would be performed when the program is executed by a real computer. The term “operational” was first used in this sense by Scott (1970), though the concept itself had already been used in specifications of the LISP programming language (McCarthy, 1960).

Operational semantics can be divided into two main approaches. Small-step semantics describes how a program is executed in *individual* steps, either as steps taken by some abstract machine or by showing how a program *reduces* to some other program. Big-step semantics, also known as *natural* semantics, describes the final result

of a program's execution, typically by inductively defining a judgement "program p in environment E produces result r". Small-step semantics were initially developed by Plotkin (2004), while big-step semantics were initially developed by Kahn (1987). The terms "small-step" and "big-step" are also due to Kahn.

In an operational setting, equality of programs is generally taken to be a simple syntactic notion. Unlike in denotational semantics, equivalent programs are not equal, making simple equality much less useful. Therefore we require a notion of equivalence between programs that is separate from equality. For this purpose we have the concept of *bisimulation*, a notion of equivalence between systems first developed by Park (1981). Two systems are said to be bisimilar if there is a relation between the states of the systems such that related states have related successor states. To put it more formally, if the first system is in state p and the second system is in related state q, and the first system can transition to state p' producing output a, the second system must be able to transition to some state q' by producing the same output a, and p' must be related to q'. Furthermore, this condition must also hold if the two systems are exchanged. The notion of bisimulation can be tailored to suit various types of systems and semantics, but all have the same general form.

Bisimulation is best suited to small-step semantics, as it involves considering two programs evaluating in lockstep. When dealing with big-step semantics, we can use the alternative notion of *observational equivalence*, as introduced by Hennessy and Milner (1980). The idea here is that if an external observer would not be able to distinguish two programs, then they are equivalent. Put more formally, if for all valid contexts $\mathbb{C}$ the programs $\mathbb{C}[p]$ and $\mathbb{C}[q]$ have indistinguishable results, then p and q are equivalent. The exact nature of the observations possible can vary, but in many cases it is enough to just consider termination, as we can often choose $\mathbb{C}[x]$ of the form "if some particular property holds of x then terminate, otherwise loop".

Related to observational equivalence is the proof technique of *coinduction* (Coquand, 1993; Turner, 1995), which comes from bisimulation. Dual to the more familiar

technique of induction, coinduction allows us to reason about infinite data by treating observationally equivalent data structures as identical. One specific case of coinduction is *guarded* coinduction, which allows use of the "coinductive hypothesis" so long as the use of the hypothesis is "guarded" by a constructor. For example, we can use guarded coinduction to prove $map\ f \circ map\ g = map\ (f \circ g)$ for infinite streams, despite the fact that infinite streams offer no base case:

$$
\begin{aligned}
& map\ f\ (map\ g\ (x : xs)) \\
= \quad & \{ \text{definition of } map \} \\
& map\ f\ (g\ x : map\ g\ xs) \\
= \quad & \{ \text{definition of } map \} \\
& f\ (g\ x) : map\ f\ (map\ g\ xs) \\
= \quad & \{ \text{coinductive hypothesis} \} \\
& f\ (g\ x) : map\ (f \circ g)\ xs \\
= \quad & \{ \text{definition of } (\circ) \} \\
& (f \circ g)\ x : map\ (f \circ g)\ xs \\
= \quad & \{ \text{definition of } map \} \\
& map\ (f \circ g)\ (x : xs)
\end{aligned}
$$

Note that the "coinductive hypothesis" step only substitutes terms that appear as the stream argument to the constructor $(:)$. This is what we mean by a "guarded" use of the hypothesis.

This apparently circular proof can be justified by appealing to an induction proof based on *finite approximations*. In the case of infinite streams, the nth *approximation* of a stream xs (written $approx\ n\ xs$) is the stream consisting of the first n elements of xs followed by the undefined stream $\bot$. Any observable behaviour of a program that operates on streams must occur after only finitely many steps, and so there must be some n such that the program only depends on the first n elements of xs. In other words, because the program can only look at finitely many places of its input before producing an output (or else it is non-terminating), there must be some n such that

if $approx\ n\ xs = approx\ n\ ys$, then the program cannot distinguish xs and ys. Therefore, we can conclude:

$$\forall\, n\,.\, approx\ n\ xs = approx\ n\ ys \quad \Rightarrow \quad xs \sim ys$$

where $\sim$ is observational equivalence. This implication is known as the *approximation lemma* (Bird, 1998).

We can use this idea to convert our coinductive proof above to an induction on n. The base case is trivial, as $approx\ 0\ xs = \bot$ for any xs. The inductive case is:

$$\begin{aligned}
& take\ (n+1)\ (map\ f\ (map\ g\ (x : xs))) \\
= \quad & \{\text{ coinductive proof up to the application of the hypothesis }\} \\
& take\ (n+1)\ (f\ (g\ x) : map\ f\ (map\ g\ xs)) \\
= \quad & \{\text{ definition of } take\ \} \\
& f\ (g\ x) : take\ n\ (map\ f\ (map\ g\ xs)) \\
= \quad & \{\text{ inductive hypothesis }\} \\
& f\ (g\ x) : take\ n\ (map\ (f \circ g)\ xs) \\
= \quad & \{\text{ definition of } take\ \} \\
& take\ (n+1)\ (f\ (g\ x) : map\ (f \circ g)\ xs) \\
= \quad & \{\text{ coinductive proof after the application of the hypothesis }\} \\
& take\ (n+1)\ (map\ (f \circ g)\ (x : xs))
\end{aligned}$$

Since all finite approximations of $map\ f\ (map\ g\ xs)$ and $map\ (f \circ g)\ xs$ are equal, no program can distinguish them, so they must be observationally equivalent. Therefore it is safe to treat them as equal.

The approximation lemma can be generalised to a wide class of algebraic data types (Hutton and Gibbons, 2001). Therefore, the proof technique of guarded coinduction can be similarly generalised.

3.1.3 Categorical Semantics

Category theory, first introduced by Eilenberg and Mac Lane (1945), is the study of *categories*, a general kind of mathematical structure. A category consists of a collection of objects, along with a collection of arrows, each arrow being associated with both a source and a target object. Furthermore, arrows can be composed so long as the source of one arrow is the target of the other, and this composition is associative. Finally, each object is equipped with an *identity* arrow that serves as the unit of composition. This machinery abstracts from the common mathematical pattern of having a kind of structure (for example, monoids) equipped with a notion of when one structure can be transformed into another.

Category theory enables us to have notions of structures like products, sums and exponentials that are agnostic in the kind of object in question. For example, the cartesian product of sets, direct product of groups, greatest lower bound of poset elements and tuple types of many programming languages are all examples of the categorical product. This high level of abstraction category theory provides has earned it the nickname "generalised abstract nonsense". Despite this moniker, category theory provides a very useful toolkit for many branches of mathematics and computer science. Typically, categorical approaches to program semantics will use objects to model datatypes and arrows to model either terms or functions.

Category theory neatly characterises the models of the simply-typed lambda calculus as being exactly the *cartesian closed* categories (Lambek, 1980), categories that are closed under both products and exponentials. It can also be shown that complete pointed partial orders (as mentioned above) form a cartesian closed category (Barendregt, 1984). However, as the simply-typed lambda calculus is total, it also has models that do not give meaning to general recursion, for example the sets and functions of standard set theory.

As well as providing a useful perspective on the simply-typed lambda calculus, category theory can also be used to give a meaning to recursive data types with *ini-*

tial algebras and *final coalgebras*. Typically, initial algebras correspond to finite data structures and final coalgebras correspond to potentially infinite data structures. This model of data types comes with a natural notion of *fold* and *unfold* operators, which break down and build up recursive data respectively. This approach was originally developed by Hagino (1987), and subsequently by Malcolm (1990).

Categorical work on initial algebras and final coalgebras was applied to the category of complete pointed partial orders by Meijer, Fokkinga, and Paterson (1991). In this setting, initial algebras and final coalgebras coincide, allowing folds and unfolds to be used at the same types. This allows us to model computations that pass around infinite data structures, which can make sense in lazy languages such as Haskell.

Chapter 4

Worker/Wrapper for Fix

"I subscribe to the myth that an artist's creativity comes from torment. Once that's fixed, what do you draw on?"

— David Byrne

4.1 Motivation

This chapter explains the basics of a denotational semantics based on *complete pointed partial orders* (CPPOs), along with a version of the worker/wrapper theory for this setting. CPPOs are a commonly-used way of giving meaning to programs written in both typed and untyped lazy functional languages, coming with a very natural notion of what it means for a function to be strict and a straightforward semantics for general recursion. The approach to the presentation of the worker/wrapper transformation given in this chapter is originally due to Sculthorpe and Hutton (2014).

4.2 Background: Least Fixed Point Semantics

Least fixed point semantics give meaning to recursive programs and types using complete pointed partial orders. This section provides an overview of the theory.

4.2.1 Complete Pointed Partial Orders

A *directed set* is a nonempty set D where every pair of elements $x, y \in D$ has an upper bound that is also in D. A poset is a *complete partial order* (CPO) when every directed subset has a least upper bound — in other words, when there is a least y such that $x \leqslant y$ for all elements x of the set. A complete partial order is *pointed* (a *CPPO*) if there is a least element. By convention, we call this least element $\bot$.

Functions between CPPOs are called *continuous* if they map directed sets to directed sets while preserving their least upper bounds. It follows from this definition that all continuous functions are monotone. In the case that f preserves the least element $\bot$ then we also say that f is *strict*.

If A is a CPPO, then a continuous function $f : A \to A$ has a *least fixed point*. A least fixed point of f is an element x of A that satisfies the following two properties:

- Fixed point property: $f\ x = x$
- Least prefix point property: for all y such that $f\ y \leqslant y$, we have $x \leqslant y$.

It follows from these two properties that least fixed points are unique. We denote the least fixed point of f as $\mathit{fix}\ f$.

CPPOs can be used as the basis of a denotational semantics for lazy functional languages, where the types are represented by CPPOs and the functions are continuous functions between those CPPOs. The bottom element $\bot$ represents a value whose computation never terminates, and strictness for continuous functions therefore corresponds precisely to strictness in the functional programming sense.

4.2.2 Fixed Point Induction, Fusion and Rolling

In order to develop our theory, we will need some rules and proof techniques to apply. We need three rules in particular, namely: fixed point induction, fixed point fusion and the rolling rule.

A *chain* is a possibly infinite sequence of elements of an order such that for each adjacent pair x_1, x_2 in the sequence, we have $x_1 \leqslant x_2$. As a chain is a special case of a directed set, all chains in CPPOs have least upper bounds. A predicate P is *chain-complete* if whenever it holds for all elements of a chain, it holds for the least upper bound of the chain. Generally predicates based on equalities between terms will be chain-complete. For any chain-complete predicate P, we have the following *fixed point induction* rule (Winskel, 1993):

$$P\ \bot \wedge (\forall x.P\ x \Rightarrow P\ (f\ x)) \Rightarrow P\ (\mathit{fix}\ f)$$

We can use fixed point induction to prove a number of useful properties, including the *fixed point fusion* rule (Meijer et al., 1991). Fusion allows us to combine a function with a fixed point to make another fixed point:

$$h\ (\mathit{fix}\ f) = \mathit{fix}\ g \quad \Leftarrow \quad h \circ f = g \circ h \wedge h\ \mathit{strict}$$

To prove this via fixed point induction, we let our predicate P be $P\ (x, y) \Leftrightarrow h\ x = y$. Starting with the base case we reason as follows:

$$\begin{array}{ll} & P\ \bot \\ \Leftrightarrow & \quad \{\text{ the bottom of a product order is } (\bot, \bot)\ \} \\ & P\ (\bot, \bot) \\ \Leftrightarrow & \quad \{\text{ definition of } P\ \} \\ & h\ \bot = \bot \\ \Leftrightarrow & \\ & h\ \mathit{strict} \end{array}$$

The inductive case is also fairly straightforward:

$$\begin{array}{ll} & P\ (f\ x, g\ y) \\ \Leftrightarrow & \quad \{\text{ definition of } P\ \} \\ & h\ (f\ x) = g\ y \end{array}$$

$\Leftrightarrow$ $\{ h \circ f = g \circ h \}$

$g\ (h\ x) = g\ y$

$\Leftarrow$ $\{ g$ is a function $\}$

$h\ x = y$

$\Leftrightarrow$ { definition of P }

$P\ (x,y)$

Noting that the fixed point of $\lambda(x,y) \rightarrow (f\ x, g\ y)$ is $(\mathit{fix}\ f, \mathit{fix}\ g)$, we can thus conclude $h\ (\mathit{fix}\ f) = \mathit{fix}\ g)$ by fixed point induction.

Finally, the *rolling rule* allows us "roll out" an f from $\mathit{fix}\ (f \circ g)$ to make $f\ (\mathit{fix}\ (g \circ f))$. This can be viewed as a special case of fusion where $f\ (\mathit{fix}\ (g \circ f))$ can be fused to make $\mathit{fix}\ (f \circ g)$. However, this would include a side condition that f is strict. With careful reasoning, we can eliminate this requirement. We prove this by antisymmetry, proving first that one side is less than the other, and then vice-versa. Firstly:

$\mathit{fix}\ (f \circ g) \leqslant f\ (\mathit{fix}\ (g \circ f))$

$\Leftarrow$ { least prefix point property }

$(f \circ g)\ (f\ (\mathit{fix}\ (g \circ f))) \leqslant f\ (\mathit{fix}\ (g \circ f))$

$\Leftrightarrow$ { expanding $\circ$ }

$f\ (g\ (f\ (\mathit{fix}\ (g \circ f)))) \leqslant f\ (\mathit{fix}\ (g \circ f))$

$\Leftrightarrow$ { contracting $\circ$ }

$f\ ((g \circ f)\ (\mathit{fix}\ (g \circ f))) \leqslant f\ (\mathit{fix}\ (g \circ f))$

$\Leftrightarrow$ { fixed point property }

$f\ (\mathit{fix}\ (g \circ f)) \leqslant f\ (\mathit{fix}\ (g \circ f))$

$\Leftrightarrow$ { reflexivity }

True

Having proved this, we can now complete the proof as follows:

$f\ (\mathit{fix}\ (g \circ f)) \leqslant \mathit{fix}\ (f \circ g)$

$\Leftrightarrow$ { fixed point property }

Given functions

$$abs : B \to A \quad f : A \to A$$
$$rep : A \to B \quad g : B \to B$$

satisfying one of the assumptions

$$\text{(A)} \quad abs \circ rep = id$$
$$\text{(B)} \quad abs \circ rep \circ f = f$$
$$\text{(C)} \quad fix\ (abs \circ rep \circ f) = fix\ f$$

If any one of the following conditions holds:

$$\text{(1)} \quad g = rep \circ f \circ abs \qquad \text{(1}\beta\text{)} \quad fix\ g = fix\ (rep \circ f \circ abs)$$
$$\text{(2)} \quad g \circ rep = rep \circ f \wedge rep\ strict \qquad \text{(2}\beta\text{)} \quad fix\ g = rep\ (fix\ f)$$
$$\text{(3)} \quad abs \circ g = f \circ abs$$

then we have the factorisation:

$$fix\ f = abs\ (fix\ g)$$

Figure 4.1: The Worker/Wrapper Theorem for Least Fixed Points

$$\begin{aligned}
& f\ (fix\ (g \circ f)) \leqslant (f \circ g)\ (fix\ (f \circ g)) \\
\Leftrightarrow \quad & \{ \text{expanding} \circ \} \\
& f\ (fix\ (g \circ f)) \leqslant f\ (g\ (fix\ (f \circ g))) \\
\Leftarrow \quad & \{ \text{monotonicity} \} \\
& fix\ (g \circ f) \leqslant g\ (fix\ (f \circ g)) \\
\Leftrightarrow \quad & \{ \text{above result, swapping } f \text{ and } g \} \\
& True
\end{aligned}$$

4.3 A Worker/Wrapper Theorem for Least Fixed Points

Suppose we have a program of type A that is written as $fix\ f$ for some function $f : A \to A$, and we wish to optimise this function by applying the worker/wrapper transformation. To do this, we need a new type B along with functions abs and rep that allow B

to faithfully represent elements of A: recall that the function *rep represents* elements of A in B while the function *abs abstracts* back to the original type A.

In order to ensure that the representation of A in B is faithful, we require a condition on *abs* and *rep*. The obvious requirement is that $abs \circ rep = id$, but there may also be weaker conditions that suffice.

Given this, we want a specification of the new worker program. We can assume that this will be of the form $fix\ g$ for some $g : B \to B$, so it is enough to have a specification of g. The function *abs* can then be used as the wrapper to convert from the new worker's type B back to A.

These requirements are satisfied by the *Worker/Wrapper Theorem for Least Fixed Points*, which can be seen in Figure 4.1. Assumptions (A), (B) and (C) capture the required relationship between *abs* and *rep*, while the five numbered conditions can be used as specifications for g.

4.3.1 Discussion

The variety of preconditions of this theorem bears some explaining. Assumption (A), the strongest of the three letter assumptions, is simply the requirement that *rep* give a faithful (in other words, reversible) representation of A in B. However, in some situations this may be too strong a requirement, and hence we have the alternate assumptions (B) and (C). Assumption (B) can be interpreted as the requirement that *rep* need a faithful representation for values that are possible results of f. Assumption (C) is similar, but adds a recursive context.

The numbered conditions are also varied. Condition (1) gives a direct definition of the function g, and so at first this might seem like the obvious choice, but this definition will often require a nontrivial amount of simplification to offer an improvement in efficiency. Conditions (2) and (3) are *almost* dual, providing ways to derive the function g by "pushing" either *rep* or *abs* through the definition of f. The β conditions are weakened versions of conditions (1) and (2), introducing a recursive context. This

added recursive context can make the conditions harder to prove, but these conditions do serve an important role in simplifying the proof, as will be seen below.

The assumptions and conditions of the theorem are not independent. In fact, they have the following implications between them:

- The assumptions form a hierarchy (A) $\Rightarrow$ (B) $\Rightarrow$ (C). Assumption (A) can be weakened to (B) by composing both sides with f, and (B) can be further weakened to (C) by applying fix to both sides.
- Both conditions (1) and (2) imply their respective β versions. Condition (1) can be weakened to (1β) by applying fix to both sides, while (2) implies (2β) by fixed point fusion.
- Under assumption (C) (and hence under any assumption), both β conditions are equivalent. This can be proved as follows:

$$
\begin{aligned}
& fix\ g = rep\ (fix\ f) \\
\Leftrightarrow \quad & \{\ \text{fixed point property}\ \} \\
& fix\ g = rep\ (f\ (fix\ f)) \\
\Leftrightarrow \quad & \{\ \text{(C)}\ \} \\
& fix\ g = rep\ (f\ (fix\ (abs \circ rep \circ f))) \\
\Leftrightarrow \quad & \{\ \text{rolling rule}\ \} \\
& fix\ g = fix\ (rep \circ f \circ abs)
\end{aligned}
$$

4.4 Proof

Because of the relationships between the preconditions of the theorem, it is enough to prove the theorem for the weakest assumption (C) and the two weakest conditions (1β) and (3). The case for (1β) is straightforward:

$$
\begin{aligned}
& fix\ f \\
= \quad & \{\ \text{(C)}\ \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathit{fix}\ (\mathit{abs} \circ \mathit{rep} \circ f) \\
= \quad & \{ \text{rolling rule} \} \\
& \mathit{abs}\ (\mathit{fix}\ (\mathit{rep} \circ f \circ \mathit{abs})) \\
= \quad & \{ (1\beta) \} \\
& \mathit{abs}\ (\mathit{fix}\ g)
\end{aligned}
$$

Now we must tackle the case of condition (3). Under (A), $\mathit{abs}\ \bot \leqslant \mathit{abs}\ (\mathit{rep}\ \bot) = \bot$, so abs is strict and (3) implies the result by fusion. However, the proof for the weaker condition (C) is more involved. We prove the result by antisymmetry. Showing $\mathit{fix}\ f \leqslant \mathit{abs}\ (\mathit{fix}\ g)$ is fairly simple:

$$
\begin{aligned}
& \mathit{fix}\ f \leqslant \mathit{abs}\ (\mathit{fix}\ g) \\
\Leftarrow \quad & \{ \text{least prefix point} \} \\
& f\ (\mathit{abs}\ (\mathit{fix}\ g)) \leqslant \mathit{abs}\ (\mathit{fix}\ g) \\
\Leftrightarrow \quad & \{ (3) \} \\
& \mathit{abs}\ (g\ (\mathit{fix}\ g)) \leqslant \mathit{abs}\ (\mathit{fix}\ g) \\
\Leftrightarrow \quad & \{ \text{fixed point property} \} \\
& \mathit{abs}\ (\mathit{fix}\ g) \leqslant \mathit{abs}\ (\mathit{fix}\ g) \\
\Leftrightarrow \quad & \{ \text{reflexivity} \} \\
& \mathit{True}
\end{aligned}
$$

To show the other direction, we must use fixed point induction. Letting $P\ y = \mathit{abs}\ y \leqslant \mathit{fix}\ f$, we have that our desired result $\mathit{abs}\ (\mathit{fix}\ g) \leqslant \mathit{fix}\ f$ is equivalent to $P\ (\mathit{fix}\ g)$, which we prove by induction. Firstly, the base case:

$$
\begin{aligned}
& P\ \bot \\
\Leftrightarrow \quad & \{ \text{definition of } P \} \\
& \mathit{abs}\ \bot \leqslant \mathit{fix}\ f \\
\Leftrightarrow \quad & \{ (C) \} \\
& \mathit{abs}\ \bot \leqslant \mathit{fix}\ (\mathit{abs} \circ \mathit{rep} \circ f)
\end{aligned}
$$

$\Leftrightarrow$ { rolling rule }

$abs\ \bot \leqslant abs\ (fix\ (rep \circ f \circ abs))$

$\Leftarrow$ { monotonicity }

$\bot \leqslant fix\ (rep \circ f \circ abs)$

$\Leftrightarrow$ { bottom element }

$True$

Then the inductive step:

$P\ (g\ y)$

$\Leftrightarrow$ { definition of P }

$abs\ (g\ y) \leqslant fix\ f$

$\Leftrightarrow$ { (3) }

$f\ (abs\ y) \leqslant fix\ f$

$\Leftrightarrow$ { fixed point property }

$f\ (abs\ y) \leqslant f\ (fix\ f)$

$\Leftarrow$ { monotonicity }

$abs\ y \leqslant fix\ f$

$\Leftrightarrow$ { definition of P }

$P\ y$

Finally, we note that in the case of a non-strict *abs*, (3) alone is not enough for the result of the theorem to hold — a counterexample for this case and a number of others can be found in Sculthorpe and Hutton (2014).

4.5 Example: Fast Reverse

Recall the example of the *reverse* function from Section 2.1.

$reverse :: [a] \to [a]$

$reverse\ [\,] \quad = [\,]$

$$reverse\ (x : xs) = reverse\ xs +\!\!+ [x]$$

If we wish to optimise this using our least fixed point-based theory, we must first transform the program into the form $fix\ f$ for some function f. We do this simply by abstracting out the recursive calls in the definition of *reverse*:

$$\begin{aligned}
&reverse = fix\ f \\
&\quad \textbf{where} \\
&\qquad f :: ([a] \to [a]) \to ([a] \to [a]) \\
&\qquad f\ rev\ [\,] \qquad = [\,] \\
&\qquad f\ rev\ (x : xs) = rev\ xs +\!\!+ [x]
\end{aligned}$$

Now, whenever before a recursive call to *reverse* is made, the same call is instead made to the function parameter of f. In this way, least fixed point semantics can give a meaning to directly recursive definitions.

The change of representation we want to apply is based on *difference lists*, due to Hughes (1986). This scheme represents a list xs by the function takes another list ys and returns $xs +\!\!+ ys$. We have the following two conversion functions:

$$\begin{aligned}
&toDiffList :: [a] \to ([a] \to [a]) \\
&toDiffList\ xs = (xs +\!\!+)
\end{aligned}$$

$$\begin{aligned}
&fromDiffList :: ([a] \to [a]) \to [a] \\
&fromDiffList\ func = func\ [\,]
\end{aligned}$$

It is easy to show that *fromDiffList* is a left inverse of *toDiffList*.

$$\begin{aligned}
&\quad fromDiffList\ (toDiffList\ xs) \\
&= \quad \{\text{ definitions }\} \\
&\quad (xs +\!\!+)\ [\,] \\
&= \quad \{\text{ operator sections }\}
\end{aligned}$$

$$
\begin{aligned}
& xs +\!\!+ [\,] \\
= \quad & \{ [\,] \text{ is the identity of } +\!\!+ \} \\
& xs
\end{aligned}
$$

This change of representation works on lists. However, *reverse* is a not a list, but a function that *returns* a list. Therefore, our *abs* and *rep* must act on functions, converting the return value of the function to and from the difference list representation.

$$
\begin{aligned}
& abs :: ([a] \to ([a] \to [a])) \to ([a] \to [a]) \\
& abs\ func = fromDiffList \circ func
\end{aligned}
$$

$$
\begin{aligned}
& rep :: ([a] \to [a]) \to ([a] \to ([a] \to [a])) \\
& rep\ func = toDiffList \circ func
\end{aligned}
$$

Our A type is therefore $[a] \to [a]$, the type of *reverse*, while our B type is $[a] \to ([a] \to [a])$. It is straightforward to show that assumption (A) is satisfied:

$$
\begin{aligned}
& abs\ (rep\ func) \\
= \quad & \{ \text{definitions} \} \\
& fromDiffList \circ toDiffList \circ func \\
= \quad & \{ \text{above proof} \} \\
& func
\end{aligned}
$$

Now that we have our *abs* and *rep* satisfying assumption (A), we must find a function g that satisfies one of the numbered conditions. In this case, it turns out to be easiest to use (2) as a specification. Firstly, we show that *rep* is strict:

$$
\begin{aligned}
& rep\ \bot \\
= \quad & \{ \text{definition of } rep \} \\
& toDiffList \circ \bot \\
= \quad & \{ \text{expanding} \circ \}
\end{aligned}
$$

$$
\begin{aligned}
& \lambda xs \to \mathit{toDiffList}\ (\bot\ xs) \\
= \quad & \{\ \bot\ x = \bot \text{ for all } x\ \} \\
& \lambda xs \to \mathit{toDiffList}\ \bot \\
= \quad & \{\ \text{definition of } \mathit{toDiffList}\ \} \\
& \lambda xs \to (\bot +\!\!+) \\
= \quad & \{\ \text{operator sections}\ \} \\
& \lambda xs\ ys \to \bot +\!\!+\ ys \\
= \quad & \{\ +\!\!+ \text{ is strict in its first argument}\ \} \\
& \lambda xs\ ys \to \bot \\
= \quad & \{\ \text{functions that are everywhere } \bot \text{ are equivalent to } \bot\ \} \\
& \bot
\end{aligned}
$$

Now, we must calculate g such that $\mathit{rep} \circ f = g \circ \mathit{rep}$. If we fully apply both sides of this equation, the calculation is fairly straightforward. The type of $\mathit{rep} \circ f$ is $([a] \to [a]) \to [a] \to ([a] \to [a])$, so the first argument is a function and the second argument is a list. We proceed by case analysis on the list argument. Firstly, we consider the case where the argument is $[\,]$:

$$
\begin{aligned}
& (\mathit{rep} \circ f)\ \mathit{rev}\ [\,] \\
= \quad & \{\ \text{definition of } \mathit{rep}\ \} \\
& \mathit{toDiffList}\ (f\ \mathit{rev}\ [\,]) \\
= \quad & \{\ \text{definition of } f\ \} \\
& \mathit{toDiffList}\ [\,] \\
= \quad & \{\ \text{definition of } \mathit{toDiffList}\ \} \\
& ([\,] +\!\!+) \\
= \quad & \{\ [\,] \text{ is the identity of } +\!\!+\ \} \\
& \mathit{id}
\end{aligned}
$$

To satisfy the equation, we need a definition of g such that

$$(g \circ \mathit{rep})\ \mathit{rev}\ [\,] = g\ (\mathit{rep}\ \mathit{rev})\ [\,] = \mathit{id}$$

We take $g\ r\ [\,] = id$. Now we consider the case where the input is $x : xs$:

$$
\begin{aligned}
& (rep \circ f))\ rev\ (x : xs) \\
= \quad & \{ \text{ definition of } rep \} \\
& toDiffList\ (f\ rev\ (x : xs)) \\
= \quad & \{ \text{ definition of } f \} \\
& toDiffList\ (rev\ xs +\!\!+ [x]) \\
= \quad & \{ \text{ definition of } toDiffList \} \\
& ((rev\ xs +\!\!+ [x])+\!\!+) \\
= \quad & \{ \text{ associativity of } +\!\!+ \text{ and properties of operator sections } \} \\
& (rev\ xs+\!\!+) \circ ([x]+\!\!+) \\
= \quad & \{ \text{ definition of } +\!\!+ \} \\
& (rev\ xs+\!\!+) \circ (x:)
\end{aligned}
$$

We need this to equal $g\ (rep\ rev)\ (x : xs)$. Starting from this side, we reason:

$$
\begin{aligned}
& g\ (rep\ rev)\ (x : xs) \\
= \quad & \{ \text{ definition of } rep \} \\
& g\ (toDiffList \circ rev)\ (x : xs) \\
= \quad & \{ \text{ definition of } toDiffList \} \\
& g\ (\lambda ys \to (rev\ ys+\!\!+))\ (x : xs)
\end{aligned}
$$

We can thus make the two sides equal by letting $g\ r\ (x : xs) = r\ xs \circ (x:)$. Thus we have derived the following definition of g:

$$
\begin{aligned}
g\ r\ [\,] &= id \\
g\ r\ (x : xs) &= r\ xs \circ (x:)
\end{aligned}
$$

This gives us the following optimised definition of *reverse*:

$$
\begin{aligned}
& reverse = abs\ (fix\ g) \\
& \quad \textbf{where}
\end{aligned}
$$

$$
\begin{aligned}
g\ r\ [\,] &= id \\
g\ r\ (x : xs) &= r\ xs \circ (x:)
\end{aligned}
$$

Rewriting this using direct recursion and expanding out the definition of abs, we get:

$$
\begin{aligned}
&reverse\ xs = rev\ xs\ [\,] \\
&\quad \textbf{where} \\
&\quad\quad rev\ [\,] = id \\
&\quad\quad rev\ (x : xs) = rev\ xs \circ (x:)
\end{aligned}
$$

Finally, if we η-expand our definition of rev and simplify, we obtain:

$$
\begin{aligned}
&reverse\ xs = rev\ xs\ [\,] \\
&\quad \textbf{where} \\
&\quad\quad rev\ [\,]\ ys = ys \\
&\quad\quad rev\ (x : xs)\ ys = rev\ xs\ (x : ys)
\end{aligned}
$$

The resulting program is precisely the optimised definition from section 2.1, modulo some renaming.

4.6 Conclusion

This chapter introduced the notion of CPPOs, demonstrated how these could be used to give a meaning to recursive programs and presented the worker/wrapper transformation for such programs, along with a correctness theorem. This theory had a correctness theorem with a wide range of potential preconditions, any one of which could be used to calculate an optimised program.

The format of the central theorem of this chapter will be repeated throughout this thesis, as other worker/wrapper theories will tend to follow the same patterns. However, as will be shown later on, the task of unifying these theories is not as straightforward as one might expect.

Chapter 5

Worker/Wrapper for Unfold

"The opposite of a correct statement is a false statement. But the opposite of a profound truth may well be another profound truth."

— Neils Bohr

5.1 Motivation

The previous chapter demonstrated a version of the worker/wrapper transformation for least fixed points, built on the framework of complete pointed partial orders. However, this approach has some limitations, two of which we shall outline here.

Firstly, the setting of complete pointed partial orders assumes a *partial* programming language, where functions can fail to return. Programming languages with totality checkers such as Idris (Brady, 2011) are becoming increasingly popular, and while we could simply ignore the partial aspects of our framework, it would clearly be a disservice not to explore the possibilities of a theory for total programs.

Secondly, least fixed points are a model of *unstructured* recursion. We might wish to make use of the reasoning tools given to us by more structured patterns, such as folds, paramorphisms or unfolds. Not only are these patterns of recursion available in languages without general recursion, the specificity can give us results that do not hold in the general case.

In this chapter, we present a worker/wrapper transformation for *unfolds*, based on the categorical concept of final coalgebras. An unfold is a function which starts from a single seed value and builds up a data structure, using recursive calls to make subterms of the structure. This pattern of building subterms with recursive calls is known as *co*recursion. For example, consider the Haskell type of infinite streams:

$$\mathbf{data}\ \mathit{Stream}\ a = a : \mathit{Stream}\ a$$

The unfold pattern for this type can be captured by the following function:

$$\begin{aligned}&\mathit{unfold} :: (a \to b) \to (a \to a) \to a \to \mathit{Stream}\ b\\&\mathit{unfold}\ h\ t\ x = h\ x : \mathit{unfold}\ h\ t\ (t\ x)\end{aligned}$$

Not only are unfolds useful in general, the language of final coalgebras allows us to use "infinite" data in a total setting by treating the data as lazy (i.e. produced on demand). Furthermore, coalgebras can also be viewed as *behaviours* of a *system*, which means that theories about them can also be applied in more operational contexts.

5.2 Background: Worker/Wrapper for folds

As unfolds and final coalgebras are dual to folds and initial algebras, it is informative to see the worker/wrapper transformation as it was developed for folds. Given a functor $\mathsf{F} : \mathcal{C} \to \mathcal{C}$, an F-*algebra* is a pair (A, f) of object A in $\mathcal{C}$ and an arrow $f : \mathsf{F}\ A \to A$. A *homomorphism* between algebras (A, f) and (B, g) is an arrow $h : A \to B$ satisfying the property $h \circ f = g \circ \mathsf{F}\ h$. This property is captured by the following commutative diagram:

$$\begin{array}{ccc}\mathsf{F}\ A & \xrightarrow{\mathsf{F}\ h} & \mathsf{F}\ B\\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g}\\ A & \xrightarrow[h]{} & B\end{array}$$

The *initial* algebra, denoted $(\mu\mathsf{F}, in)$, is the unique F-algebra from which there is a unique homomorphism to any F-algebra. We write the unique homomorphism from

$(\mu \mathsf{F}, in)$ to (A, f) as $fold\ f$. The initial algebra can be thought of as a type of terms with syntax defined by F, and $fold\ f$ a function that recursively evaluates a term by applying f to the evaluation of its subterms.

As the operator $fold$ captures a notion of structural recursion, it makes sense to ask whether programs written in terms of $fold$ can be factorised in the same way as programs written in terms of fix, into a worker program at a different type and a wrapper program handling the conversion. The answer is yes: $fold$ has fusion and rolling rules similar to those for fix, so a similar worker/wrapper development is possible.

The resulting *worker/wrapper transformation for folds* was initially developed by Hutton et al. (2010). This work produced the insight of having multiple numeric conditions: the original version of the transformation for fix had the assumptions (A–C) but only condition (1). The subsequent development by Sculthorpe and Hutton (2014) integrated both versions of the theory, creating fix and $fold$ theories with the full breadth of conditions that we consider in this thesis. We present the central theorem from this latter work in Figure 5.1.

5.3 Background: Final Coalgebras

Suppose that we fix a category $\mathcal{C}$ and a functor $\mathsf{F} : \mathcal{C} \to \mathcal{C}$ on this category. Then an F-*coalgebra* is a pair (A, f) consisting of an object A along with an arrow $f : A \to \mathsf{F}\ A$. We often omit the object A as it is implicit in the type of f. A *homomorphism* between coalgebras $f : A \to \mathsf{F}\ A$ and $g : B \to \mathsf{F}\ B$ is an arrow $h : A \to B$ satisfying the property $\mathsf{F}\ h \circ f = g \circ h$. This property is captured by the following commutative diagram:

$$\begin{array}{ccc} A & \xrightarrow{h} & B \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle g} \\ \mathsf{F}\ A & \xrightarrow[\mathsf{F}\ h]{} & \mathsf{F}\ B \end{array}$$

Given functions

$$abs : B \to A \qquad f : \mathsf{F}\ A \to A$$
$$rep : A \to B \qquad g : \mathsf{F}\ B \to B$$

satisfying one of the assumptions

$$\text{(A)} \quad abs \circ rep = id$$
$$\text{(B)} \quad abs \circ rep \circ f = f$$
$$\text{(C)} \quad fold\ (abs \circ rep \circ f) = fold\ f$$

If any one of the following conditions holds:

$$\text{(1)} \quad g = rep \circ f \circ \mathsf{F}\ abs \qquad \text{(1}\beta\text{)} \quad fold\ g = fold\ (rep \circ f \circ \mathsf{F}\ abs)$$
$$\text{(2)} \quad g \circ \mathsf{F}\ rep = rep \circ f \qquad \text{(2}\beta\text{)} \quad fold\ g = rep \circ fold\ f$$
$$\text{(3)} \quad abs \circ g = f \circ \mathsf{F}\ abs$$

then we have the factorisation:

$$fold\ f = abs \circ fold\ g$$

Figure 5.1: The Worker/Wrapper Theorem for Folds (Sculthorpe and Hutton, 2014)

Intuitively, a coalgebra $f : A \to \mathsf{F}\ A$ can be thought of as giving a *behaviour* to elements of A, where the possible behaviours are specified by the functor F. For example, if we define $\mathsf{F}\ X = 1 + X$ on the category **Set** of sets and total functions, then a coalgebra $f : A \to 1 + A$ is the transition function of a state machine in which each element of A is either a terminating state or has a single successor. In turn, a homomorphism corresponds to a behaviour-preserving mapping, in the sense that if we first apply the homomorphism h and then the target behaviour captured by g, we obtain the same result as if we apply the source behaviour captured by f and then map h across the A components of the result.

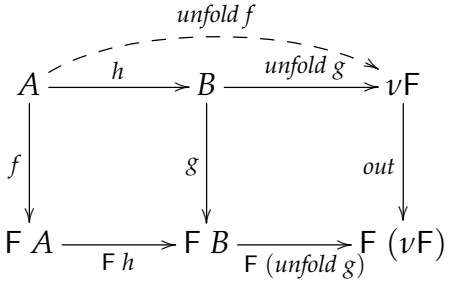
A *final* coalgebra, denoted $(\nu\mathsf{F}, out)$, is an F-coalgebra to which any other coalgebra has a unique homomorphism. If a final coalgebra exists, it is unique up to isomorphism. Given a coalgebra $f : A \to \mathsf{F}\ A$, the unique homomorphism from f to the final coalgebra out is denoted $unfold\ f :: A \to \nu\mathsf{F}$. This *uniqueness property* can be captured by the following equivalence:

$$h = unfold\ f \quad \Leftrightarrow \quad \mathsf{F}\ h \circ f = out \circ h$$

We also have a fusion rule for $unfold$. Given arrows $f : A \to \mathsf{F}\ A$, $g : B \to \mathsf{F}\ B$ and $h : A \to B$ we have the following implication:

$$\begin{array}{ll} & unfold\ g \circ h = unfold\ f \\ \Leftarrow & \\ & \mathsf{F}\ h \circ f = g \circ h \end{array}$$

The proof of this rule can be conveniently captured by the following commutative diagram:



The left square commutes by assumption while the right square commutes because $unfold\ g$ is a coalgebra homomorphism. Therefore, the outer rectangle commutes, meaning that $unfold\ g \circ h$ is a homomorphism from f to out. Finally, because homomorphisms to the final coalgebra out are unique and $unfold\ f$ is also such a homomorphism, the result $unfold\ g \circ h = unfold\ f$ holds.

A corollary of fusion is the *rolling rule* for $unfold$, which echoes the rolling rule for least fixed points. The proof is simple:

$$
\begin{array}{ll}
 & unfold\ (\mathsf{F}\ f \circ g) \circ f = unfold\ (g \circ f) \\
\Leftarrow & \quad \{\ \text{fusion}\ \} \\
 & \mathsf{F}\ f \circ g \circ f = \mathsf{F}\ f \circ g \circ h \\
\Leftarrow & \quad \{\ \text{identity}\ \} \\
 & True
\end{array}
$$

We illustrate the above concepts with a concrete example. Consider the functor $\mathsf{F}\ X = Nat \times X$ on the category **Set**. This functor has a final coalgebra consisting of the set $Stream\ Nat$ of streams of natural numbers together with the function $\langle head, tail \rangle : Stream\ Nat \to Nat \times Stream$ that combines the stream destructors $head : Stream\ Nat \to Nat$ and $tail : Stream\ Nat \to Stream\ Nat$. Given any set A and functions $h : A \to Nat$ and $t : A \to A$, the function $unfold\langle h, t \rangle : A \to Stream\ Nat$ is uniquely defined by the two equations

$$
\begin{aligned}
head \circ unfold\langle h, t \rangle &= h \\
tail \ \circ unfold\langle h, t \rangle &= unfold\langle h, t \rangle \circ t
\end{aligned}
$$

which are equivalent to following more intuitive definition we had earlier.

For a more thorough treatment of the subject of coalgebras, see Rutten (2000).

Given functions

$$abs : B \to A \qquad f : A \to \mathsf{F}\ A$$
$$rep : A \to B \qquad g : B \to \mathsf{F}\ B$$

satisfying one of the assumptions

(A) $abs \circ rep = id$
(B) $f \circ abs \circ rep = f$
(C) $unfold\ (f \circ abs \circ rep) = unfold\ f$

If any one of the following conditions holds:

(1) $g = \mathsf{F}\ rep \circ f \circ abs$ (1β) $unfold\ g = unfold\ (\mathsf{F}\ rep \circ f \circ abs)$
(2) $\mathsf{F}\ abs \circ g = f \circ abs$ (2β) $unfold\ g = unfold\ f \circ abs$
(3) $g \circ rep = \mathsf{F}\ rep \circ f$

then we have the factorisation:

$$unfold\ f = unfold\ g \circ rep$$

Figure 5.2: The Worker/Wrapper Theorem for Unfolds

5.4 The Worker/Wrapper Theorem for Unfolds

Given all this background, we can now present the *worker/wrapper theorem for unfolds*, as seen in Figure 5.2. (The proof of this theorem is given in section 5.7.) Given a program of the form $unfold\ f : A \to \nu\mathsf{F}$ and conversion functions $abs : B \to A$ and $rep : A \to B$ we can use the theorem to factorise the original unfold to $unfold\ g \circ rep$, with the theorem itself providing us with a specification for g.

5.5 Discussion

This theorem bears a great degree of similarity to the theorem for least fixed points in the previous chapter. However, there are also a number of key differences between this theorem and the fix theorem. It is also worth comparing the theorem here to the theorem given in Figure 5.1 for folds, which are dual to unfolds. In this section, we

make some observations on how our unfold theorem compares to these two theorems.

Firstly, assumption (B) in the fix and fold theorems, i.e. $abs \circ rep \circ f = f$, can be interpreted as "for any x in the range of f, $abs\ (rep\ x) = x$". This can therefore be proven by reasoning only about such x. When applying the theorem for unfolds, this kind of simple reasoning for assumption (B) is not possible, as f is now applied last rather than first and hence cannot be factored out of the proof in this way.

Secondly, condition (2) in the fix case (that rep is strict and $rep \circ f = g \circ rep$) and the similar condition (2) in the fold theorem allow g to depend on a precondition set up by rep. If such a precondition is desired for the unfold case, condition (3) must be used. This has important implications for use of this theorem as a basis for optimisation, as we will often derive g based on a specification given by one of the conditions.

Thirdly, while condition (2) in the fix case had a strictness side-condition, in this case there are no side conditions. This ought to make the theorem easier to apply. The fold theorem also has no side conditions.

Finally, we note that proving (C), (1β) or (2β) for the fix or fold case usually requires induction. To prove the corresponding properties for the unfold case will usually require the less widely-understood technique of coinduction. These properties may therefore turn out to be less useful in the unfold case for practical purposes, despite only requiring a technique of comparable complexity. If we want to use this theory to avoid coinduction altogether, assumption (C) and the β conditions are not applicable. The following section offers a way around this problem in the case of (C).

5.6 Refining Assumption (C)

As it stands, assumption (C) is expressed as an equality between two corecursive programs defined using $unfold$, and hence may be non-trivial to prove. However, we can derive an equivalent assumption that may be easier to prove in practice:

$$
\begin{array}{ll}
& \mathit{unfold}\ f = \mathit{unfold}\ (f \circ \mathit{abs} \circ \mathit{rep}) \\
\Leftrightarrow & \quad \{ \text{uniqueness property of } \mathit{unfold}\ (f \circ \mathit{abs} \circ \mathit{rep}) \} \\
& \mathit{out} \circ \mathit{unfold}\ f = \mathsf{F}\ (\mathit{unfold}\ f) \circ f \circ \mathit{abs} \circ \mathit{rep} \\
\Leftrightarrow & \quad \{ \mathit{unfold}\ f \text{ is a coalgebra homomorphism} \} \\
& \mathit{out} \circ \mathit{unfold}\ f = \mathit{out} \circ \mathit{unfold}\ f \circ \mathit{abs} \circ \mathit{rep} \\
\Leftrightarrow & \quad \{ \mathit{out} \text{ is an isomorphism} \} \\
& \mathit{unfold}\ f = \mathit{unfold}\ f \circ \mathit{abs} \circ \mathit{rep}
\end{array}
$$

We denote this equivalent version of assumption (C) as (C'). As this new assumption concerns only the conversions *abs* and *rep* along with the original program $\mathit{unfold}\ f$, it may be provable simply from the original program's correctness properties.

This proof dualises easily for the fold theorem due to Sculthorpe and Hutton. Unfortunately however, we cannot apply this kind of trick to the theorem for least fixed points as least fixed points lack an equivalent to the unfolds' uniqueness property.

5.7 Proof

The assumptions and conditions have the same interrelationships as they did in the previous chapter, namely that (A) $\Rightarrow$ (B) $\Rightarrow$ (C); that (1) and (2) each imply their respective β conditions; and that under assumption (C) the two β conditions are equivalent. Unlike the fix case, however, there are no strictness side-conditions to work around: (3) implies the result simply by fusion, so no involved proof-by-induction is needed. It therefore suffices to prove the theorem for assumption (C) and condition (1β). The proof of this follows the same structure as that for fix, being centred around an application of the rolling rule:

$$
\begin{array}{ll}
& \mathit{unfold}\ f \\
= & \quad \{ \text{(C)} \} \\
& \mathit{unfold}\ (f \circ \mathit{abs} \circ \mathit{rep})
\end{array}
$$

$$
\begin{aligned}
&= \quad \{ \text{ rolling rule } \} \\
&\quad \mathit{unfold}\ (\mathit{rep} \circ f \circ \mathit{abs}) \circ \mathit{rep} \\
&= \quad \{ (1\beta) \} \\
&\quad \mathit{unfold}\ g \circ \mathit{rep}
\end{aligned}
$$

Noting that (C) can be equivalently phrased as the (C') condition we derived above, there is an even simpler proof available. Assuming (C') and (2β), we have:

$$
\begin{aligned}
&\quad \mathit{unfold}\ f \\
&= \quad \{ (C') \} \\
&\quad \mathit{unfold}\ f \circ \mathit{abs} \circ \mathit{rep} \\
&= \quad \{ (2\beta) \} \\
&\quad \mathit{unfold}\ g \circ \mathit{rep}
\end{aligned}
$$

However, it was not trivial to prove that (C') is equivalent to (C). Essentially, we have moved work from one place to another.

5.8 Unifying

Now that we have two different versions of the worker/wrapper theory with similar theorems and similar proofs, it would be logical to ask if we can unify them. If we only wish to deal with unfolds in the category of CPPOs, then we can simply rewrite our unfolds using general recursion. A similar approach to this was used by Sculthorpe and Hutton to partially unite the fix theory with a theory for *folds*, the dual of unfolds (Sculthorpe and Hutton, 2014). This resulted in a fold theory with an additional strictness side-condition. This reflects the fact that the folds in the category of CPPOs are not true folds: they are not the unique homomorphism from the initial algebra, merely the *least* homomorphism from a *weakly* initial algebra[1]. The category of CPPOs *does* have true unfolds, so in this case there is no need for any side conditions.

[1] By weakly initial, we mean that there is a (not necessarily unique) homomorphism from it to any other algebra.

Unfortunately, going further than this presents a number of obstacles. Firstly, a unified theory would have to explain why there is a strictness side condition in the case of least fixed-points, but not for unfolds. Secondly, the fact that there is an equivalent formulation of (C') here but not in the previous theory suggests that the theories are in some way unequal, which is something we would have to address. Finally, we noted above that condition (2) of the fix theory seems to serve the same purpose of condition (3) here, suggesting that conditions (2) and (3) have swapped roles. However, condition (3) has no β condition. This suggests that our theories so far lack some symmetry that will be needed if we are to truly unify them. In fact this is exactly the case, as the next chapter will show.

5.9 Examples

Before we move on to the next chapter, however, we will demonstrate the use of the worker/wrapper transformation with some worked examples. Firstly, we look at the problem of tabulating the results of a function. Secondly, we look at the problem of cycling a list. In both of these cases the examples take place in a total setting, namely the category $\mathbf{Set}$ of sets and total functions.

5.9.1 Tabulating a Function

The function *tabulate* takes a function on the natural numbers and produces the infinite stream of its results:

$$tabulate\ f = [f\ 0, f\ 1, f\ 2, f\ 3, ...$$

A fairly straightforward definition of this is as follows:

$$\begin{aligned} &tabulate :: (Nat \to a) \to Stream\ a \\ &tabulate\ f = f\ 0 : tabulate\ (f \circ (+1)) \end{aligned}$$

Unfortunately, this definition repeats a great deal of work, as each result of the output stream is the result of evaluating a term of the form $(f \circ (+1) \circ (+1) \circ ... \circ (+1))\ 0$. Essentially, this definition obscures the opportunity to reuse work in calculating the input of f. Noting that $Stream\ a$ is the final coalgebra of the functor F $X = a \times X$, we can rewrite *tabulate* as an unfold:

$$
\begin{aligned}
&tabulate = unfold\ h\ t \\
&\quad \textbf{where}\ h\ f = f\ 0 \\
&\qquad\qquad t\ f = f \circ (+1)
\end{aligned}
$$

To improve this, we want to use the worker/wrapper transformation to factorise the repeated composition of f with $(+1)$. We can do this by keeping the most recent input value to f alongside it, changing our original type $Nat \to a$ to the tuple type $(Nat \to a, Nat)$. We convert between these types with the following *abs* and *rep* functions.

$$
\begin{aligned}
&rep :: (Nat \to a) \to (Nat \to a, Nat) \\
&rep\ f = (f, 0)
\end{aligned}
$$

$$
\begin{aligned}
&abs :: (Nat \to a, Nat) \to (Nat \to a) \\
&abs\ (f, n) = f \circ (+n)
\end{aligned}
$$

Assumption (A) can be proven easily:

$$
\begin{aligned}
&\ \ abs\ (rep\ f) \\
&= \quad \{\ \text{definition of } rep\ \} \\
&\ \ abs\ (f, 0) \\
&= \quad \{\ \text{definition of } abs\ \} \\
&\ \ f \circ (+0) \\
&= \quad \{\ (+0) = id\ \} \\
&\ \ f
\end{aligned}
$$

Now we choose a condition to use as the specification for our transformed program. In this instance we use condition (2). We can specialise this condition as follows:

$$
\begin{aligned}
& \mathsf{F}\ abs \circ g = f \circ abs \\
\Leftrightarrow \quad & \{\ \text{let} f = \langle h, t\rangle, g = \langle h', t'\rangle\ \} \\
& \mathsf{F}\ abs \circ \langle h', t'\rangle = \langle h, t\rangle \circ abs \\
\Leftrightarrow \quad & \{\ \text{definition of F, products}\ \} \\
& \langle h', abs \circ t'\rangle = \langle h \circ abs, t \circ abs\rangle \\
\Leftrightarrow \quad & \{\ \text{separating components}\ \} \\
& h' = h \circ abs \ \ \wedge \ \ abs \circ t' = t \circ abs
\end{aligned}
$$

Therefore, if we can find h' and t' that match this specification, then we will be able to use $unfold\ \ h'\ \ t' \circ rep$ as our factorised version of $tabulate$. Starting with the left equation, we calculate the following definition of h':

$$h'\ (f, n) = (h \circ abs)\ (f, n) = h\ (f \circ (+n)) = f\ (0 + n) = f\ n$$

The right equation is slightly more involved. We start with the right hand side:

$$
\begin{aligned}
& (t \circ abs)\ (f, n) \\
= \quad & \{\ \text{definition of } abs\ \} \\
& t\ (f \circ (+n)) \\
= \quad & \{\ \text{definition of } t\ \} \\
& (f \circ (+n)) \circ (+1) \\
= \quad & \{\ \text{associativity of function composition}\ \} \\
& f \circ ((+n) \circ (+1)) \\
= \quad & \{\ \text{arithmetic}\ \} \\
& f \circ (+(n + 1)) \\
= \quad & \{\ \text{definition of } abs\ \} \\
& abs\ (f, n + 1) \\
= \quad & \{\ \text{letting } t'\ (f, n) = (f, n + 1)\ \} \\
& (abs \circ t')\ (f, n)
\end{aligned}
$$

Therefore, we have that $tabulate = unfold\ h'\ t' \circ rep$. Writing this out in full, we get:

$$
\begin{aligned}
&\mathit{tabulate}\ f = \mathit{work}\ (f, 0) \\
&\quad \textbf{where}\ \mathit{work}\ (f, n) = f\ n : \mathit{work}\ (f, n+1)
\end{aligned}
$$

In this version, repeated work is avoided by keeping the input to f alongside the function itself, and modifying this rather than the function. This saves an amount of time linear in the number of elements of the output stream that is requested.

5.9.2 Cycling a List

The function *cycle* takes a non-empty finite list and produces the stream consisting of repetitions of that list. For example:

$$\mathit{cycle}\ [1,2,3] = [1,2,3,1,2,3,1,2,3,...$$

One possible definition for *cycle* is as follows, in which we write $[a]^+$ for the type of non-empty lists of type a:

$$
\begin{aligned}
&\mathit{cycle} :: [a]^+ \to \mathit{Stream}\ a \\
&\mathit{cycle}\ (x : xs) = x : \mathit{cycle}\ (xs +\!\!+ [x])
\end{aligned}
$$

However, this definition is inefficient, as the append operator $+\!\!+$ takes linear time in the length of the input list. We can rewrite *cycle* as the unfold:

$$
\begin{aligned}
&\mathit{cycle} = \mathit{unfold}\ h\ t \\
&\quad \textbf{where}\ h\ xs = \mathit{head}\ xs \\
&\qquad\qquad t\ xs\ = \mathit{tail}\ xs +\!\!+ [\mathit{head}\ xs]
\end{aligned}
$$

The idea we shall apply to improve the performance of *cycle* is to combine several $+\!\!+$ operations into one, thus reducing the average cost. To achieve this, we create a new representation where the original list of type $[a]^+$ is augmented with a (possibly empty) list of elements that have been added to the end. We keep this second list in reverse order so that appending a single element is a constant-time operation. The *rep* and *abs* functions are as follows:

$$rep :: [a]^+ \to ([a]^+, [a])$$
$$rep\ xs = (xs, [\,])$$

$$abs :: ([a]^+, [a]) \to [a]^+$$
$$abs\ (xs, ys) = xs +\!\!+ reverse\ ys$$

Given these definitions it is easy to verify assumption (A):

$$\begin{aligned}
&abs\ (rep\ xs)\\
=\quad &\{\text{ definition of } rep\ \}\\
&abs\ (xs, [\,])\\
=\quad &\{\text{ definition of } abs\ \}\\
&xs +\!\!+ reverse\ [\,]\\
=\quad &\{\text{ definition of } reverse\ \}\\
&xs +\!\!+ [\,]\\
=\quad &\{\ [\,] \text{ is unit of } +\!\!+\ \}\\
&xs
\end{aligned}$$

For this example we take condition (2), i.e. F $abs \circ g = f \circ abs$, as our specification of g, once again specialising to the two conditions $h \circ abs = h'$ and $t \circ abs = abs \circ t'$. From this we can calculate h' and t' separately. First we calculate h':

$$\begin{aligned}
&h'\ (xs, ys)\\
=\quad &\{\text{ specification }\}\\
&h\ (abs\ (xs, ys))\\
=\quad &\{\text{ definition of } abs\ \}\\
&h\ (xs +\!\!+ reverse\ ys))\\
=\quad &\{\text{ definition of } h\ \}\\
&head\ (xs +\!\!+ reverse\ ys)\\
=\quad &\{\ xs \text{ is nonempty }\}\\
&head\ xs
\end{aligned}$$

Now we calculate a definition for t'. Starting from the specification $abs \circ t' = t \circ abs$, we calculate as follows:

$$
\begin{aligned}
& t\ (abs\ (xs, ys)) \\
= \quad & \{ \text{definition of } abs \} \\
& t\ (xs +\!\!+ reverse\ ys)) \\
= \quad & \{ \text{case analysis} \} \\
& \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad \to t\ ([x] \qquad +\!\!+ reverse\ ys) \\
& \quad (x : xs') \to t\ ((x : xs') +\!\!+ reverse\ ys) \\
= \quad & \{ \text{definition of } +\!\!+ \} \\
& \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad \to t\ (x : ([\,] \ +\!\!+ reverse\ ys)) \\
& \quad (x : xs') \to t\ (x : (xs' +\!\!+ reverse\ ys)) \\
= \quad & \{ \text{definition of } t \} \\
& \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad \to [\,] \ +\!\!+ reverse\ ys +\!\!+ [x] \\
& \quad (x : xs') \to xs' +\!\!+ reverse\ ys +\!\!+ [x] \\
= \quad & \{ \text{definition of } reverse, +\!\!+ \} \\
& \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad \to \qquad reverse\ (x : ys) \\
& \quad (x : xs') \to xs' +\!\!+ reverse\ (x : ys) \\
= \quad & \{ \text{definition of } abs \} \\
& \mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad \to abs\ (reverse\ (x : ys), [\,]) \\
& \quad (x : xs') \to abs\ (xs', x : ys) \\
= \quad & \{ \text{pulling } abs \text{ out of cases} \} \\
& abs\ (\mathbf{case}\ xs\ \mathbf{of} \\
& \quad [x] \qquad \to (reverse\ (x : ys), [\,]) \\
\end{aligned}
$$

$$
\begin{aligned}
&(x : xs') \to (xs', x : ys) \\
&)
\end{aligned}
$$

Hence, t' can be defined as follows:

$$
\begin{aligned}
t'\ ([x], ys) &= (reverse\ (x : ys), [\,]) \\
t'\ (x : xs, ys) &= (xs, x : ys)
\end{aligned}
$$

In conclusion, by applying our worker-wrapper theorem, we have calculated a factorised version of *cycle*

$$
\begin{aligned}
&cycle = unfold\ h'\ t' \circ rep \\
&\quad \textbf{where}\ h'\ (xs, ys) = head\ xs \\
&\qquad\qquad t'\ ([x], ys) = (reverse\ (x : ys), [\,]) \\
&\qquad\qquad t'\ (x : xs, ys) = (xs, x : ys)
\end{aligned}
$$

which can be written directly as

$$
\begin{aligned}
&cycle = cycle' \circ rep \\
&\quad \textbf{where} \\
&\qquad cycle'\ ([x], ys) = x : cycle'\ (reverse\ (x : ys), [\,]) \\
&\qquad cycle'\ (x : xs, ys) = x : cycle'\ (xs, x : ys)
\end{aligned}
$$

This version only performs a *reverse* operation once for every cycle of the input list, so the average cost to produce a single element is now constant. We believe that this kind of optimisation — in which costly operations are delayed and combined into a single operation — will be a common use of our theory.

5.10 Conclusion

This chapter introduced the categorical notion of *coalgebra*, and showed that *final* coalgebras could be used to give a semantics to potentially infinite data structures.

This comes with a suitable categorical definition of *unfolds*, a recursion scheme that builds up large data structures by using recursive calls to compute subterms. We then presented a worker/wrapper transformation for unfolds complete with a correctness theorem, and demonstrated its similarity to the previous worker/wrapper transformation for least fixed points.

So far we have two similar worker/wrapper theories, but no satisfactory way to unify them. The problem of unification will be the focus of the next chapter.

Chapter 6

Generalising

"The English have all the material requisites for the revolution. What they lack is the spirit of generalization and revolutionary ardour."
— Karl Marx, Importance and Weakness of English Labour

6.1 Introduction

Thus far, all instances of the worker/wrapper transformation have centered on an application of a *rolling* or *fusion* rule, properties that allow functions to be moved into and out of a recursive context. Variants of these rules exist for a wide class of recursion operators, so this seems a natural starting point for developing a generic theory. As it turns out, the appropriate generalisations of rolling and fusion rules are the categorical notions of weak and strong *dinaturality*.

Dinaturality arises in category theory as a generalisation of the notion of natural transformations, families of morphisms with a commutativity property. For example, the natural transformation $reverse : [A] \to [A]$ that reverses the order of a list satisfies the property $reverse \circ map\ f = map\ f \circ reverse$, where map applies a function to each element of a list. In a simple categorical semantics where objects correspond to types and arrows correspond to functions, natural transformations correspond to

the familiar notion of *parametric polymorphism*, and their commutativity properties arise as free theorems (Wadler, 1989) for the types of polymorphic functions.

However, as a model of polymorphism, natural transformations have a significant limitation: their source and target must be *functors*. This means that polymorphic functions where the type variable appears *negatively* in either the source or target, for example $\mathit{fix} : (A \to A) \to A$, cannot be defined as natural transformations. For this reason, the concept of naturality is sometimes generalised to *di*naturality and strong dinaturality. To put it in categorical terms, dinatural transformations generalise natural transformations to the case where the source and target of the transformation may have *mixed* variance.

It has been widely observed (Bainbridge, Freyd, Scedrov, and Scott, 1990; Uustalu, 2010) that there is a relationship between (strong) dinaturality and parametricity, the property from which free theorems follow, although the exact details of this relationship are unclear. It is known that parametricity entails dinaturality (Plotkin and Abadi, 1993) and for certain types even strong dinaturality (Ghani, Uustalu, and Vene, 2004). These observations suggest the possibility of a *categorical* notion of parametricity that sits between the two, but as of yet there is no such property that we are aware of. The situation is not helped by the wide variety of models of parametricity that have been developed. For the purposes of this chapter, we assume that all recursion operators of interest are strongly dinatural; in practice, we are not aware of any such operators in common use that do not satisfy this assumption.

This chapter develops a generic version of the worker/wrapper transformation, applicable to a wide class of recursion operators, with a correctness theorem based around the categorical notion of strong dinaturality. In this way we establish strong dinaturality as the *essence* of the worker/wrapper transformation, and obtain a general theory that is applicable to a wide class of recursion operators. Not only do all existing worker/wrapper correctness theories arise as instances of the general theory, but the theory can also be used to derive new instances, including a theory for

monadic fixed points and an interesting degenerate case for arrow fixed points.

6.2 The Essence of Worker/Wrapper

Recall the worker/wrapper theory for least fixed points. In this case, we have the original program written as a fixed point of a function $f : A \to A$, and wish to derive a new function $g : B \to B$, such that the following equation holds:

$$fix\ f = abs\ (fix\ g)$$

Here, $fix\ f$ is the original program of type A, while abs is the wrapper and $fix\ g$ is the worker of type B. The original formulation of worker/wrapper (Gill and Hutton, 2009) used the following proof of correctness:

$$\begin{array}{ll} & fix\ f \\ = & \quad \{\, abs \circ rep = id \,\} \\ & fix\ (abs \circ rep \circ f) \\ = & \quad \{\, \text{rolling rule} \,\} \\ & abs\ (fix\ (rep \circ f \circ abs) \\ = & \quad \{\, \text{define } g = rep \circ f \circ abs \,\} \\ & abs\ (fix\ g) \end{array}$$

This proof gives us a direct definition for the new function g, to which standard techniques can then be used to 'fuse together' the functions in the definition for g to give a more efficient implementation for the worker program $fix\ g$.

The worker/wrapper theory for folds (Hutton et al., 2010) has a proof of correctness based on fusion. Adapting this proof to the fix case, we obtain:

$$\begin{array}{ll} & fix\ f = abs\ (fix\ g) \\ \Leftrightarrow & \quad \{\, abs \circ rep = id \,\} \\ & abs\ (rep\ (fix\ f)) = abs\ (fix\ g) \end{array}$$

$\Leftarrow$ { unapplying abs }

$rep\ (fix\ f) = fix\ g$

$\Leftarrow$ { fusion, assuming rep is strict }

$rep \circ f = g \circ rep$

This proof gives a *specification* for the new function g in terms of the given functions rep and f, from which the aim is then to calculate an implementation. It also appears as a subproof of the complete proof for the worker/wrapper transformation for fixed points that is presented in Sculthorpe and Hutton (2014).

Both of the above proofs essentially have only one non-trivial step. In the first proof, this is the use of the rolling rule. In the second proof, it is the use of fusion.

6.2.1 Generalising

There are several reasons why we would like to generalise the least fixed point presentation to a wider range of settings. Firstly, the full power of the fixed-point operator fix is not always available to the programmer. This is becoming increasingly the case as the popularity of dependently typed languages such as Agda and Coq increases, as these languages tend to have totality requirements that preclude the use of general recursion. Secondly, the general recursion that is provided by the use of fixed-points is unstructured, and other recursion operators such as folds and unfolds can be significantly easier to reason with in practice. Finally, the least fixed points presentation is tied to the framework of complete pointed partial orders, preventing us from applying the theory to languages where this semantic model does not apply.

The uniform presentation of two previously unrelated worker/wrapper theories due to Sculthorpe and Hutton (2014) is promising but also somewhat unsatisfactory, as it does little to explain *why* such a uniform presentation is possible, only demonstrating that it is. Nevertheless, this work provided a vital stepping-stone toward the general theory we present in this chapter.

Earlier in this section, we noted that both proofs center on an application of either the rolling rule or fusion. For this reason, we believe it is appropriate to view these rules as the "essence" of the worker/wrapper transformation. Thus, to generalise the worker/wrapper transformation, the first step is to generalise these rules. In this case, the appropriate generalisation of the rolling rule is the category-theoretic notion of *dinaturality* (Mulry, 1990). The fusion rule can be similarly generalised to the notion of *strong* dinaturality (Uustalu, 2010).

6.3 Dinaturality and Strong Dinaturality

Now we shall explain the concepts of dinaturality and strong dinaturality, including their relationship with the rolling rule and fusion.

Firstly, we recall the notion of a natural transformation. For two functors $\mathsf{F}, \mathsf{G} : \mathcal{C} \to \mathcal{D}$, a family of arrows $\alpha_A : \mathsf{F}\ A \to \mathsf{G}\ A$ is a *natural transformation* if for any $f : A \to B$, the following diagram commutes:

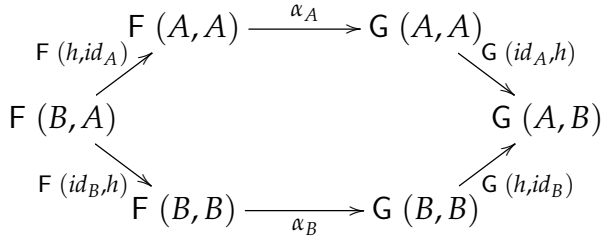
$$\begin{array}{ccc} \mathsf{F}\ A & \xrightarrow{\alpha_A} & \mathsf{G}\ A \\ {\scriptstyle \mathsf{F} f}\downarrow & & \downarrow{\scriptstyle \mathsf{G} f} \\ \mathsf{F}\ B & \xrightarrow{\alpha_B} & \mathsf{G}\ B \end{array}$$

This diagram is a *coherence* property, essentially requiring that each of the α_A, α_B "do the same thing", independent of the choice of the particular A. In this way, natural transformations provide a categorical notion of parametric polymorphism.

However, some polymorphic operators, such as $fix : (A \to A) \to A$ cannot be expressed as natural transformations. This is because natural transformations require both their source and target to be *functors*, whereas in the case of fix the source type $A \to A$ is not functorial because A appears in a "negative" position. It is natural to ask whether there is a categorical notion that captures these operators as well, where the source and target may not be functors. The notion of *di*naturality was developed

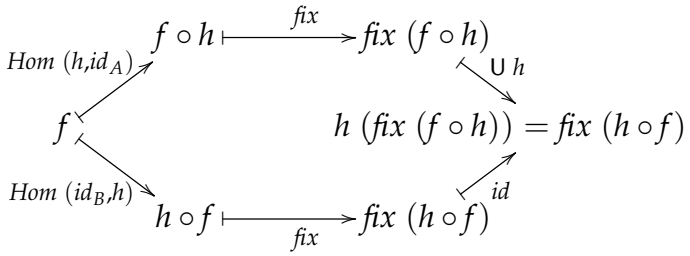
for precisely such cases.

For two functors $\mathsf{F}, \mathsf{G} : \mathcal{C}^{op} \times \mathcal{C} \to \mathcal{D}$, a family of arrows $\alpha_A : \mathsf{F}\ (A, A) \to \mathsf{G}\ (A, A)$ is a *dinatural transformation* if for any $h : A \to B$, the following diagram commutes:



For *fix*, this property exactly captures the rolling rule. To see this, take $\mathcal{C} = \mathbf{Cpo}$ (the category of complete pointed partial orders and continuous functions) and $\mathcal{D} = \mathbf{Set}$ (the category of sets and total functions). We let $\mathsf{F}\ (X, Y) = Hom\ (X, Y)$ and $\mathsf{G}\ (X, Y) = \mathsf{U}\ Y$ where U is the forgetful functor that takes a CPPO and returns its underlying set, and assume a (continuous) function $f \in \mathsf{F}\ (B, A) = B \to A$. Then *fix* will be a dinatural transformation that takes an element of $\mathsf{F}\ (X, Y)$ and returns one of $\mathsf{G}\ (X, Y)$, i.e. a function from $Hom\ (X, Y)$ to $\mathsf{U}\ Y$. (Actually, *fix* is a transformation in $\mathbf{Cpo}$ rather than in $\mathbf{Set}$, but we can forget this extra structure for our purposes.)

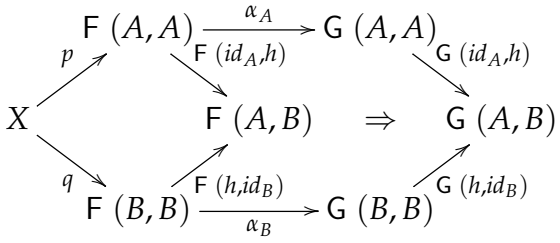
Chasing the function f around the above diagram, we obtain the rolling rule:



Note that $\mathsf{G}\ (h, id_B)$ expands simply to id_B because G ignores its contravariant argument. We can use this diagram-chasing technique to obtain rolling rules for other recursion operators such as *fold* and *unfold*. Thus we see that dinaturality can be considered a generalisation of the rolling rule.

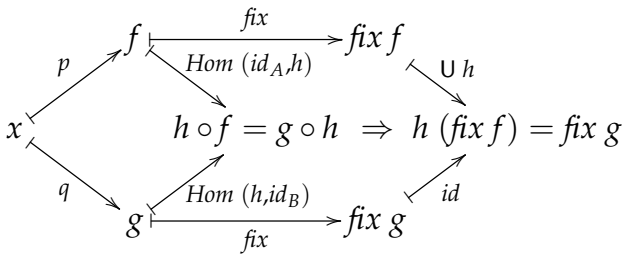
For some purposes, however, the notion of dinaturality is too weak. For exam-

ple, the composition of two dinatural transformations is not necessarily dinatural. For this reason, the stronger property of *strong* dinaturality is sometimes used. This property is captured by the following diagram, which should be read as "if the diamond on the left commutes, then the outer hexagon commutes".



If we set $X = \mathsf{F}\ (B, A)$, $p = \mathsf{F}\ (h, id_A)$ and $q = \mathsf{F}\ (id_B, h)$, then the diamond on the left commutes by functoriality and so the diagram as a whole reduces to ordinary dinaturality. Thus we confirm that strong dinaturality is indeed a stronger property.

Applying strong dinaturality to *fix* in a similar manner to previously, we see that it corresponds to a *fusion* rule. Choosing some $x : X$ and letting $p\ x = f : A \to A$ and $q\ x = g : B \to B$, we chase values around the diagram as before:



Thus, strong dinaturality in this case states that $h \circ f = g \circ h$ implies $h\ (fix\ f) = fix\ g$. The fusion rule is precisely this property, with an extra strictness condition on h. This strictness condition can be recovered by treating F and G as functors from the *strict subcategory* $\mathbf{Cpo}_\perp$ in which all arrows are strict. The functor $\mathsf{F}\ (X, Y)$ is still defined as the full function space $\mathbf{Cpo}\ \ (X, Y)$, including non-strict arrows. The mixing of strict and non-strict arrows is a little awkward, but we must do this as *fix* is useless when limited to strict arrows, as the least fixed point of any strict arrow is $\perp$.

Thus we see that while dinaturality is a generalisation of the rolling rule, *strong* dinaturality is a generalisation of fusion. Because the rolling and fusion rules are the essence of the worker/wrapper transformation, it makes sense to use (strong) dinaturality as the basis for developing a generalised theory. We develop such a theory in the following section.

6.4 Worker/Wrapper For Strong Dinaturals

Suppose we have chosen a particular programming language to work with, and let $\mathcal{C}$ be the category where the objects are types in that language and the arrows are functions from one type to another. Then a polymorphic type $\forall x \;.\; T$ where x appears in both positive and negative positions in T can be represented by a functor $\mathsf{F} : \mathcal{C}^{op} \times \mathcal{C} \to \mathbf{Set}$, where $\mathsf{F}\ (A, A)$ is the set of terms in the language of type $T[A/x]$. In turn, a *recursion operator* that takes terms of type $\mathsf{F}\ (A, A)$ and produces terms of type $\mathsf{G}\ (A, A)$ can be represented by a strong dinatural transformation from F to G. It is known that for certain types, strong dinaturality will follow from a free theorem (Ghani et al., 2004). For example, the free theorem for the typing $fix : (A \to A) \to A$ is fusion, which we showed in the previous section to be equivalent to strong dinaturality.

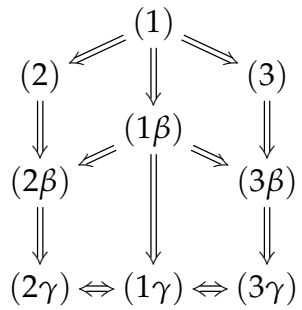
Now we present the central result of this chapter, a general categorical worker/wrapper theorem, in Figure 6.1. The data in the theorem can be interpreted as follows:

- The category $\mathcal{C}$ is a programming language.
- The objects A and B are types in the language.
- The functors F, G are type expressions with a free variable, possibly with exponentials and of mixed variance.
- The arrows $abs : B \to A$ and $rep : A \to B$ are functions in the language.

Given:

- A category $\mathcal{C}$ containing objects A and B
- Functors $\mathsf{F}, \mathsf{G} : \mathcal{C}^{op} \times \mathcal{C} \to \mathbf{Set}$
- Arrows $abs : B \to A$ and $rep : A \to B$ in $\mathcal{C}$
- The assumption $abs \circ rep = id_A$
- Elements $f \in \mathsf{F}\ (A, A)$ and $g \in \mathsf{F}\ (B, B)$
- A strong dinatural transformation $\alpha : \mathsf{F} \to \mathsf{G}$

The conditions of the theorem are related as shown in the following diagram:



If any one of the following conditions holds:

(1) $g = \mathsf{F}\ (abs, rep)\ f$
(1β) $\alpha_B\ g = \alpha_B\ (\mathsf{F}\ (abs, rep)\ f)$
(1γ) $\mathsf{G}\ (rep, abs)\ (\alpha_B\ g) = \mathsf{G}\ (rep, abs)\ (\alpha_B\ (\mathsf{F}\ (abs, rep)\ f))$

(2) $\mathsf{F}\ (rep, id)\ g = \mathsf{F}\ (id, rep)\ f$
(2β) $\mathsf{G}\ (rep, id)\ (\alpha_B\ g) = \mathsf{G}\ (id, rep)\ (\alpha_A\ f)$
(2γ) $\mathsf{G}\ (rep, abs)\ (\alpha_B\ g) = \mathsf{G}\ (id, abs \circ rep)\ (\alpha_A\ f)$

(3) $\mathsf{F}\ (id, abs)\ g = \mathsf{F}\ (abs, id)\ f$
(3β) $\mathsf{G}\ (id, abs)\ (\alpha_B\ g) = \mathsf{G}\ (abs, id)\ (\alpha_A\ f)$
(3γ) $\mathsf{G}\ (rep, abs)\ (\alpha_B\ g) = \mathsf{G}\ (abs \circ rep, id)\ (\alpha_A\ f)$

then we have the factorisation:

$$\alpha_A\ f = \mathsf{G}\ (rep, abs)\ (\alpha_B\ g)$$

Figure 6.1: The Worker/Wrapper Theorem for Strong Dinatural Transformations

- The elements $f \in \mathsf{F}\ (A, A)$ and $g \in \mathsf{F}\ (B, B)$ are terms in the language.
- The strong dinatural α is a recursion operator.

Under these interpretations, we can see that the theorem allows us to factorise a program written as $\alpha_A\ f$ into a worker program $\alpha_B\ g$ and wrapper function $\mathsf{G}\ (rep, abs)$.

The wealth of conditions in Figure 6.1 requires some explanation. Up until this point, worker/wrapper theorems in the literature had varying numbers of possible correctness conditions, ranging from just one in the original theory (Gill and Hutton, 2009) to a total of five in Sculthorpe and Hutton (2014) and previous chapters of this thesis. This variation is a result of the way previous theories were first developed separately and then unified, and all previous conditions are included in some generalised form in our presentation. The nine conditions given here were chosen to best expose the symmetries in the theory. In practical applications, one selects the condition that results in the simplest calculation for the worker program.

The conditions are related in various ways. Firstly, the (2) and (3) groups of conditions are categorically dual. This can be seen by exchanging $\mathcal{C}$ for the opposite category $\mathcal{C}^{op}$, and then swapping the roles of abs and rep. Note that the dinatural transformation is still in the same direction.

Secondly, each numeric condition (n) implies the corresponding condition $(n\beta)$, which in turn implies $(n\gamma)$. Thus the γ conditions are the weakest conditions for the theorem. These relationships can be proved as follows:

- (1) is weakened to (1β) by applying α_B to each side.
- (2) implies (2β) and (3) implies (3β) by strong dinaturality. Note that because the target of the functors is $\mathbf{Set}$, strong dinaturality can be written pointwise as

$$\begin{array}{l} \mathsf{F}\ (h, id)\ f = \mathsf{F}\ (id, h)\ g \\ \Rightarrow \\ \mathsf{G}\ (h, id)\ (\alpha_A\ f) = \mathsf{G}\ (id, h)\ (\alpha_B\ h) \end{array}$$

- (1β), (2β) and (3β) can be weakened to their corresponding γ conditions by applying G (rep, abs), G (id, abs) and G (rep, id) to each side respectively.

Thirdly, using the assumption that $abs \circ rep = id$, condition (1) implies conditions (2) and (3). The same can be said of the corresponding β conditions. In the first case this can be shown by simply applying F (rep, id) or F (id, abs) to both sides of condition (1). In the second case, one applies either G (rep, id) or G (id, abs) to both sides of (1β), and the result then follows from applying dinaturality.

Finally, using $abs \circ rep = id$ all three γ conditions are equivalent. In fact, the right hand sides are all equal. The proof that (1γ) is equivalent to (2γ) is as follows:

$$
\begin{aligned}
& \mathsf{G}\ (rep, abs)\ (\alpha_B\ (\mathsf{F}\ (abs, rep)\ f)) \\
= \quad & \{\text{ functors }\} \\
& \mathsf{G}\ (id, abs)\ (\mathsf{G}\ (rep, id)\ (\alpha_B\ (\mathsf{F}\ (id, rep)\ (\mathsf{F}\ (abs, id)\ f)))) \\
= \quad & \{\text{ dinaturality }\} \\
& \mathsf{G}\ (id, abs)\ (\mathsf{G}\ (id, rep)\ (\alpha_A\ (\mathsf{F}\ (rep, id)\ (\mathsf{F}\ (abs, id)\ f)))) \\
= \quad & \{\text{ functors }\} \\
& \mathsf{G}\ (id, abs \circ rep)\ (\alpha_A\ (\mathsf{F}\ (abs \circ rep, id)\ f)) \\
= \quad & \{ abs \circ rep = id \text{ implies } \mathsf{F}\ (abs \circ rep, id) = id \} \\
& \mathsf{G}\ (id, abs \circ rep)\ (\alpha_A\ f)
\end{aligned}
$$

The proof for (1γ) and (3γ) is dual. Thus we see that all three are equivalent.

The basic relationships between the conditions are summarised in the right-hand side of Figure 6.1. Given these relationships, it suffices to prove the theorem for one of the γ conditions. For example, it can be proved for (2γ) simply by applying the assumption $abs \circ rep = id$:

$$
\begin{aligned}
& \mathsf{G}\ (rep, abs)\ (\alpha_B\ g) \\
= \quad & \{ (2\gamma) \} \\
& \mathsf{G}\ (id, abs \circ rep)\ (\alpha_A\ f) \\
= \quad & \{ abs \circ rep = id \}
\end{aligned}
$$

$$
\begin{aligned}
& \mathsf{G}\ (id, id)\ (\alpha_A\ f) \\
= \quad & \{\, \text{functors} \,\} \\
& \alpha_A\ f
\end{aligned}
$$

We include these trivial conditions as it allows us to break up the proof as a whole into individually trivial steps.

Conditions (1), (2) and (3) are precisely the three correctness conditions given in the original worker/wrapper theory for fold (Hutton et al., 2010), while the corresponding β and γ conditions are weakenings of those conditions. The β conditions are simply the weakenings obtained by adding a recursive context, while the γ conditions are weakened further so that they are all equivalent, much like the two weakened conditions of Sculthorpe and Hutton (2014). However, those two conditions correspond here to the conditions (1β) and (2β), which in this generalised setting are not in general equivalent.

It is also worth noting that only conditions (2) and (3) rely on strong dinaturality, which is necessary for them to imply the β conditions. With all other conditions, the theorem follows from the weaker dinaturality property.

In earlier work, weaker versions of the assumption $abs \circ rep = id$ were also considered (Gill and Hutton, 2009; Sculthorpe and Hutton, 2014). For example, the proof of correctness for the original presentation of the worker/wrapper transformation in terms of least fixed points still holds if the assumption is weakened to $abs \circ rep \circ f = f$, or further to $fix\ (abs \circ rep \circ f) = fix\ f$. The lack of weaker alternative assumptions means that our new theory is not a full generalisation of the earlier work. While this is not a significant issue, it is a little unsatisfactory. In our theorem, the assumption $abs \circ rep = id$ is used four times. For each of those four uses, a different weakening can be made. The four weakened versions are as follows:

(C1) $\alpha_A\ (\mathsf{F}\ (abs \circ rep, id)\ f) = \alpha_A\ f$

(C2) $\alpha_A\ (\mathsf{F}\ (id, abs \circ rep)\ f) = \alpha_A\ f$

(C3) $\mathsf{G}\ (id, abs \circ rep)\ (\alpha_A\, f) = \alpha_A\, f$

(C4) $\mathsf{G}\ (abs \circ rep, id)\ (\alpha_A\, f) = \alpha_A\, f$

We call these assumptions (Cn), as they are related to the (C) assumptions from the literature. The first two assumptions, (C1) and (C2), are used to prove that (1γ) is equivalent to (2γ) and (3γ) respectively, while (C3) and (C4) are used to prove the result from those two same conditions. As would be expected from this, (C1) is dual to (C2), and (C3) is dual to (C4).

There is also some duality between (C1) and (C4), and between (C2) and (C3). Strengthening the assumptions by removing the application to f gives us

(C1) $\alpha_A \circ \mathsf{F}\ (abs \circ rep, id) = \alpha_A$

(C2) $\alpha_A \circ \mathsf{F}\ (id, abs \circ rep) = \alpha_A$

(C3) $\mathsf{G}\ (id, abs \circ rep) \circ \alpha_A = \alpha_A$

(C4) $\mathsf{G}\ (abs \circ rep, id) \circ \alpha_A = \alpha_A$

in which case the duality holds exactly.

Despite these relationships, we have yet to devise a single equality weaker than $abs \circ rep = id$ that implies the correctness of the generalised worker/wrapper theorem. We suspect that doing so would require additional assumptions to be made about the strong dinatural transformation α.

6.5 Examples

In this section, we demonstrate the generality of our new theory by specialising to four particular dinatural transformations. The first two such specialisations give rise to the worker/wrapper theories for fix and unfold as presented in previous chapters. The last two specialisations are new.

6.5.1 Least Fixed Points

Firstly, we shall consider the least fixed point operator, $\mathit{fix} : (A \to A) \to A$. This can be considered a dinatural transformation of type F $\to$ G if we take the following definitions for the underlying functors F and G:

$$\mathsf{F}\ (A, B) = \mathbf{Cpo}\ (A, B)$$

$$\mathsf{G}\ (A, B) = \mathsf{U}\ B$$

Where U is the forgetful functor that takes a CPPO and returns its underlying set. Recalling the discussion from Section 6.3, we note that the functors must be typed $\mathsf{F}, \mathsf{G} : \mathbf{Cpo}_{\perp}^{op} \times \mathbf{Cpo}_{\perp} \to \mathbf{Set}$ in order to obtain the correct version of the strong dinaturality property.

By instantiating the theorem from Figure 6.1 for the fix operator, we obtain the following set of preconditions:

(1) $\quad g = \mathit{rep} \circ f \circ \mathit{abs}$

(2) $\quad g \circ \mathit{rep} = \mathit{rep} \circ f$

(3) $\quad \mathit{abs} \circ g = f \circ \mathit{abs}$

(1β) $\quad \mathit{fix}\ g = \mathit{fix}\ (\mathit{rep} \circ f \circ \mathit{abs})$

(2β) $\quad \mathit{fix}\ g = \mathit{rep}\ (\mathit{fix}\ f)$

(3β) $\quad \mathit{abs}\ (\mathit{fix}\ g) = \mathit{fix}\ f$

(1γ) $\quad \mathit{abs}\ (\mathit{fix}\ g) = \mathit{abs}\ (\mathit{fix}\ (\mathit{rep} \circ f \circ \mathit{abs}))$

(2γ) $\quad \mathit{abs}\ (\mathit{fix}\ g) = \mathit{abs}\ (\mathit{rep}\ (\mathit{fix}\ f))$

(3γ) $\quad \mathit{abs}\ (\mathit{fix}\ g) = \mathit{fix}\ f$

Note that the functions abs and rep must be strict, because they are arrows in $\mathbf{Cpo}_{\perp}$. However, the hom-sets $\mathbf{Cpo}\ (A, A)$ and $\mathbf{Cpo}\ (B, B)$ are the *full* function spaces, so their respective elements f and g need not be strict. By instantiating the conclusion of the theorem we obtain the worker/wrapper factorisation $\mathit{fix}\ f = \mathit{abs}\ (\mathit{fix}\ g)$ from Gill and Hutton (2009); Sculthorpe and Hutton (2014).

As one would expect from the previous section, the first five of the preconditions correspond to the five conditions given for the least fixed point theory presented earlier int this thesis. However that theory had only one strictness requirement: for condition (2), rep must be strict to imply the conclusion. Here, we require both abs and rep to be strict for all conditions.

We can eliminate most of these strictness conditions by noting two things. Firstly, we note that the strictness of abs is guaranteed by the assumption $abs \circ rep = id$:

$$
\begin{array}{ll}
& abs\ \bot \\
\preceq & \{\text{ monotonicity }\} \\
& abs\ (rep\ \bot) \\
= & \{\, abs \circ rep = id \,\} \\
& \bot
\end{array}
$$

Secondly, by examining the proof we can see that the full power of strong dinaturality is only needed for conditions (2) and (3), and in all other cases dinaturality suffices. As there are no strictness side conditions for the rolling rule, we can also elide strictness conditions for the normal dinaturality property. As condition (3) relies on strong dinaturality being applied with abs, for which we already have strictness guaranteed, the only strictness condition remaining is the requirement that rep be strict in (2) as in the earlier paper.

6.5.2 Unfolds

Next, we consider the unfold operator. For a functor H with a final coalgebra νH, the unfold operator for the type A takes an arrow of type $A \to \mathsf{H}\ A$ and extends it to an arrow of type $A \to \nu\mathsf{H}$. That is, we have the following typing:

$$unfold : (A \to \mathsf{H}\ A) \to A \to \nu\mathsf{H}$$

As we stated in the previous chapter, one of the key properties of the unfold operator is the *fusion* law. Given arrows $f : A \to \mathsf{H}\ A, g : B \to \mathsf{H}\ B, h : A \to B$, fusion is

captured by the following implication:

$$unfold\ f \circ h = unfold\ g \quad \Leftarrow \quad f \circ h = \mathsf{H}\ h \circ g$$

This can be recast into the language of strong dinatural transformations in a fairly straightforward manner. In particular, if we define the functors $\mathsf{F}, \mathsf{G} : \mathcal{C}^{op} \times \mathcal{C} \to \mathbf{Set}$ by

$$\mathsf{F}\ (A, B) = \mathcal{C}\ (A, \mathsf{H}\ B)$$

$$\mathsf{G}\ (A, B) = \mathcal{C}\ (A, \nu\mathsf{H})$$

then the operator *unfold* is a strong dinatural transformation from F to G. The strong dinaturality property corresponds precisely to the fusion law for unfolds given above.

Instantiating the worker/wrapper theorem from Figure 6.1 in this context gives the following set of preconditions:

(1) $g = \mathsf{H}\ rep \circ f \circ abs$

(2) $g \circ rep = \mathsf{H}\ rep \circ f$

(3) $\mathsf{H}\ abs \circ g = f \circ abs$

(1β) $unfold\ g = unfold\ (\mathsf{H}\ rep \circ f \circ abs)$

(2β) $unfold\ g \circ rep = unfold\ f$

(3β) $unfold\ g = unfold\ f \circ abs$

(1γ) $unfold\ g \circ rep = unfold\ (\mathsf{H}\ rep \circ f \circ abs) \circ rep$

(2γ) $unfold\ g \circ rep = unfold\ f$

(3γ) $unfold\ g \circ rep = unfold\ f \circ abs \circ rep$

Instantiating the conclusion gives us $unfold\ f = unfold\ g \circ rep$. In this case, five of the conditions (namely (1),(2),(3),(1β) and (3β)) are identical to conditions from the theorem in the previous chapter. The roles of the (2) and (3) conditions have been swapped, explaining the exchanged roles of these conditions that we noted before.

We note that it is unnecessary to assume anything about the object $\nu\mathsf{H}$ in this presentation: strong dinaturality is sufficient to get all the necessary properties of

the unfold operator. Thus, we have generalised the unfold theory to a more general theory that can apply to “unfold-like" operators, i.e. dinatural operators with type $\forall\ A, B\ .\ (A\ \to\ \mathsf{H}\ B)\ \to\ (A\ \to\ X)$, where X does not vary in A or B. The resulting theory could be applied in settings where final coalgebras do not exist.

6.5.3 Monadic Fixed Points

Monads are a mathematical construct commonly used in programming language theory to deal with effects such as state and exceptions (Wadler, 1992). Languages like Haskell use monads to embed effectful computations into a pure language. In this context, a value of type $\mathsf{M}\ A$ for some monad M is an *effectful* computation that produces a result of type A, where the nature of the underlying effect is captured by the monad M.

Formally, a monad is a type constructor M equipped with two operations of the following types:

$$\mathit{return} : A \to \mathsf{M}\ A$$

$$\mathit{bind} \quad : \mathsf{M}\ A \to (A \to \mathsf{M}\ B) \to \mathsf{M}\ B$$

The bind operation is often written infix as $\gg\!=$. The monad operations must obey the following three *monad laws*:

$$xm \gg\!= \mathit{return} = xm$$

$$\mathit{return}\ x \gg\!= f = f\ x$$

$$(xm \gg\!= f) \gg\!= g = xm \gg\!= (\lambda x \to f\ x \gg\!= g)$$

Given these operations and properties, the type constructor M can be made into a functor by the following definition:

$$(\mathsf{M}\ f)\ xm = xm \gg\!= (\mathit{return} \circ f)$$

It is often useful to be able to perform recursive computations within a monad. For many effects, the appropriate interpretation of recursion is unclear, as there are

sometimes multiple plausible implementations with significantly different semantics. Furthermore, while one could define a uniform monadic recursion operator using fix, for most monads this results in nontermination. For these reasons, some monads come equipped with a *monadic fix* operation $mfix : (A \to \mathsf{M}\ A) \to \mathsf{M}\ A$. Monadic fix operations are required to follow a number of laws, but here we concern ourselves only with one such law, which follows from parametricity (Erkök and Launchbury, 2000). For any strict function $s : A \to B$ and functions $f : A \to \mathsf{M}\ A$, $g : B \to \mathsf{M}\ B$, we have:

$$\mathsf{M}\ s\ (mfix\ f) = mfix\ g \quad \Leftarrow \quad \mathsf{M}\ s \circ f = g \circ s$$

This property is similar to the fusion property of the ordinary fix operator. In fact, if we define functors $\mathsf{F}\ (A, B) = \mathcal{C}\ (A, \mathsf{M}\ B)$ and $\mathsf{G}\ (A, B) = \mathsf{M}\ B$, we can see that this property precisely states that $mfix$ is a strong dinatural transformation from F to G. Using this fact, we can instantiate our worker/wrapper theorem for the case of monadic fixed points.

The preconditions are listed below. Note that once again we have a strictness side condition on rep, though in this case we cannot eliminate it from conditions as we could before as we lack the necessary non-strict rolling rule property. However, once again we can ignore strictness conditions on abs.

(1) $g = \mathsf{M}\ rep \circ f \circ abs$

(2) $g \circ rep = \mathsf{M}\ rep \circ f$

(3) $\mathsf{M}\ abs \circ g = f \circ abs$

(1β) $mfix\ g = mfix\ (\mathsf{M}\ rep \circ f \circ abs)$

(2β) $mfix\ g = \mathsf{M}\ rep\ (mfix\ f)$

(3β) $\mathsf{M}\ abs\ (mfix\ g) = mfix\ f$

(1γ) $\mathsf{M}\ abs\ (mfix\ g) = \mathsf{M}\ abs\ (mfix\ (rep \circ f \circ abs))$

(2γ) $\mathsf{M}\ abs\ (mfix\ g) = \mathsf{M}\ (abs \circ rep)\ (mfix\ f)$

(3γ) $\mathsf{M}\ abs\ (mfix\ g) = mfix\ f$

Instantiating the conclusion gives the factorisation $\mathit{mfix}\ f = \mathsf{M}\ \mathit{abs}\ (\mathit{mfix}\ g)$. This theorem is more-or-less what one might expect given the similarity between mfix and the normal fix operation, but monadic recursion has not previously been studied in the context of the worker/wrapper transformation and the theorem is therefore new. It is our general theory of strong dinatural transformations that allows us to quickly and easily generate a theorem that can now be used to apply the worker/wrapper transformation to programs written using monadic recursion. Note that we used none of the monad operations and rules, relying entirely on the strong dinaturality property of mfix, so our theory requires only that M be a functor to ensure that F and G are truly functorial in both A and B.

6.5.4 Arrow Loops

Unfortunately, monads cannot capture all notions of effectful computation we may wish to use. For this reason, we may sometimes choose to use a more general framework such as *arrows* (Hughes, 2000). An arrow is a binary type constructor Arr together with three operations of the following types:

$$\mathit{arr} : (A \to B) \to \mathsf{Arr}\ A\ B$$

$$\mathit{seq} : \mathsf{Arr}\ A\ B \to \mathsf{Arr}\ B\ C \to \mathsf{Arr}\ A\ C$$

$$\mathit{second} : \mathsf{Arr}\ A\ B \to \mathsf{Arr}\ (C \times A)\ (C \times B)$$

The seq operator is typically written infix as $\ggg$. Arrows are required to obey a number of laws, which we shall not list here. However, we do note the associativity law:

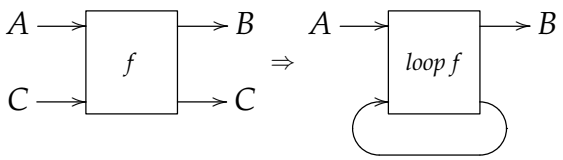
$$(f \ggg g) \ggg h = f \ggg (g \ggg h)$$

In general, arrows are a particular form of category, where the objects are the same as the underlying category of the programming language, and $\mathsf{Arr}\ A\ B$ represents the set of arrows from A to B. The operation arr is thus a functor from the underlying category of the language to the category represented by the arrow structure.

Thus far, arrows have no notion of recursion. However, some arrows provide an extra *loop* combinator (Paterson, 2001):

$$loop : \mathsf{Arr}\ (A \times C)\ (B \times C) \to \mathsf{Arr}\ A\ B$$

Intuitively, $loop$ connects one of the outputs of an arrow back into one of its inputs, as in the following picture:



Once again, loops are expected to satisfy a number of laws that we shall not list here. The arrow and loop laws imply that if $f \ggg second\ (arr\ h) = second\ (arr\ h) \ggg g$ then $loop\ f = loop\ g$, which in turn implies that $loop$ is a dinatural transformation $\mathsf{F} \to \mathsf{G}$ between the following functors:

$$\mathsf{F}\ (X, Y) = \mathsf{Arr}\ (A \times X)\ (B \times Y)$$

$$\mathsf{G}\ (X, Y) = \mathsf{Arr}\ A\ B$$

Therefore, by instantiating our worker/wrapper theorem we can conclude that, given abs and rep such that $abs \circ rep = id$ and one of the following preconditions:

(1) $g = second\ (arr\ abs) \ggg f \ggg second\ (arr\ rep)$

(2) $g \ggg second\ (arr\ rep) = second\ (arr\ abs) \ggg f$

(3) $second\ (arr\ abs) \ggg g = f \ggg second\ (arr\ rep)$

(1β) $loop\ g = loop\ (second\ (arr\ abs) \ggg f \ggg second\ (arr\ rep))$

(1γ) $loop\ g = loop\ (second\ (arr\ abs) \ggg f \ggg second\ (arr\ rep))$

then we can conclude $loop\ f = loop\ g$. (The remaining conditions (2β), (2γ), (3β) and (3γ) all amount to $loop\ f = loop\ g$, so they are not particularly useful.)

We have again instantiated our general theory to produce a novel worker/wrapper theory with very little effort, allowing the worker/wrapper transformation to be ap-

plied to programs written using the arrow loop combinator. Just as was the case for monadic recursion, our theorem is based entirely on the property of strong dinaturality, and thus does not require any of the arrow laws to hold beyond the assumption that the loop operator is strongly dinatural. Note that in this case we have a degenerate form of the conclusion where the wrapper is just the identity, because the functor G ignores its inputs.

6.6 Conclusion

We began this chapter by observing that the worker/wrapper transformation is typically centered around an application of either a rolling rule or a fusion rule. From this simple observation, we have developed a generic worker/wrapper transformation based on strong dinatural transformations that can be applied to a wide range of recursion operators. As is fitting of a generalisation, many of the proofs in this chapter follow the same basic structure as the proofs in earlier chapters. These proofs retain the same simple character as the originals, albeit in a more abstract language.

We demonstrated the broad utility of this new generic theory by instantiating it to a number of different specific recursion operators. Each instantiation provides the same variation of preconditions as was found in previous worker/wrapper theories, and thus retains the same wide applicability.

Chapter 7

Summary and Evaluation

"Let's do the time warp again!"

— Chorus of Phantoms, The Rocky Horror Show

In Part I, we presented three theories of the correctness of the worker/wrapper transformation. All three theories have some commonalities, with the structure of the proofs and theorems following roughly the same basic structure in all three cases. In two out of the three cases there were three choices available for the relationship between the functions abs and rep, and in all three there were multiple choices for the specification to use to derive the new program. On the other hand, all three differ greatly in which programs and settings they are applicable to.

The first correctness theory was based on general recursion, specifically modelling generally-recursive programs as least fixed points of functions on CPPOs. In one sense, this is a highly general theory, as any form of recursion can be implemented using general recursion. However, this restricts us to CPPO-based denotational semantics rather than broader models. Furthermore, implementing recursion schemes in this way introduces a certain amount of overhead into the theory, making the presentation less clean.

The second correctness theory was based on categorical unfolds. This gains some generality over the first theory, as it can be applied to any setting in which unfolds

exist rather than being restricted to one particular setting. On the other hand, the less general form of recursion means that the theory is only applicable to programs written in the form of an unfold, rather than any generally-recursive program as was the case before. This is less of a restriction than it may initially seem, as a great deal of programs that build data structures can be expressed as unfolds. However, this still serves as motivation to develop a theory that is general with regard to both setting and recursion scheme.

The third correctness theory was based on strong dinatural transformations. This is a generalisation of the prior theories, and we showed how it could be instantiated to give both existing and new worker/wrapper theories. The process of generalisation gave us important insights, most significantly in demonstrating that fusion and/or rolling is the core of the worker/wrapper transformation. However, this theory is not a perfect generalisation for two main reasons. Firstly, it lacks the separate (A), (B) and (C) assumptions of the prior theories. Part of the problem is that it is unclear what the appropriate generalisations of (B) and (C) would be, or even if they exist at all. Finding such generalisations would be a matter for further work. It should be noted that assumption (A) is by far the most commonly used in practice. Secondly, the way in which the strictness side conditions were obtained for the fix theory was rather ad hoc, relying on the observation that the conditions were only present for strong dinaturality, and not for ordinary dinaturality. We may need better categorical models of strictness in order to rectify this.

Part III: Improvement

Making it better

Chapter 8

Introduction

"I've got to admit, it's getting better; a little better, all the time. (It can't get no worse!)"
— The Beatles, Getting Better

In the second part of this thesis, we addressed the problem of *correctness* for the worker/wrapper transformation, proving for various frameworks that the transformation does not alter the meaning of a program. However, it is not enough to know that our transformations are correct: we must also know that they *improve* the program in some way, typically in terms of memory or time needed to execute. There is little point in an "optimisation" that doubles the running time of any program it is applied to. If we are to justify the worker/wrapper transformation as an optimisation, we must be able to make statements about these efficiency properties.

Typically, such statements are justified by empirical tests, for example benchmarks (Partain, 1992), and formal proofs of efficiency properties are rarely provided. This is in stark contrast to the case of correctness, where formal proofs are often used to the exclusion of any empirical method. Consider the Knuth aphorism: "Beware of bugs in the above code; I have only proved it correct, not tried it" (Knuth, 1977).

In this thesis we aim to redress the balance by backing up statements about efficiency with formal proofs. In this way we hope not only to justify the worker/wrapper

transformation itself, but also to promote more formal ways of justifying efficiency properties. The bulk of this chapter is given to an overview of various theories and techniques for reasoning about program efficiency.

After this chapter, the main body of this part of the thesis is divided into two chapters. The first chapter presents proofs of efficiency properties for a general-recursion-based form of the worker/wrapper transformation, using the operational *improvement theory*. The second chapter extends the generalised worker/wrapper transformation of chapter 6 to consider efficiency properties based on the categorical framework of *preorder-enriched categories*. After this, there is a short concluding chapter.

8.1 Background

For our work on program correctness, we first had to explain what we *meant* by correctness. To do this, we had to review the field of *program semantics*. Now, before we can work on the problem of program improvement, we must define what we mean by *improvement*. In this section, we shall explain the two frameworks used in this thesis for reasoning about program improvement. The first, *improvement theory à la Sands*, has its basis in operational semantics, while the second, *preorder-enriched categories*, has its basis in category theory. After this, we give a brief overview of other techniques for reasoning about efficiency.

8.1.1 Improvement Theory à la Sands

Improvement theory is an approach to reasoning about efficiency based on operational semantics. The general idea is that two terms can be compared by counting the resources each term uses in all possible contexts. Given two terms S and T, if for every context $\mathbb{C}$ we know that $\mathbb{C}[S]$ requires no more resources to evaluate than $\mathbb{C}[T]$, we say that S is an *improvement* of T. This idea can be applied to a wide range of resources, including both time and space usage.

This theory was developed initially by Sands (1991) for the particular case of the call-by-name lambda calculus. Subsequently, Moran and Sands (1999a) developed a theory for call-by-need time costs, while Gustavsson and Sands (1999, 2001) developed the corresponding theory for space usage.

While improvement theory provides the machinery needed to reason about the behaviour of call-by-need languages that are traditionally considered unpredictable, it is unfortunately limited by being tied to specific operational semantics. As a result, the conclusions that can be drawn are only valid so long as the operational semantics is a good model of the actual implementation, and so long as the model of resource usage is accurate. In particular, the theory of Sands (1991) assumes that the language implementation is based on a certain abstract machine and that counting heap lookups is a good model of time cost.

8.1.2 Preorder-Enriched Categories

Category theory offers us one fundamental way to compare arrows: by asking if they are equal or not. This makes the theory ideal for reasoning about equivalence of programs. However, if we wish to reason about other properties, we require additional structure. For this purpose, we use the machinery of *enriched* category theory (Kelly, 1982). In general, categories can be enriched over a wide variety of structures, but in this case we shall use preorders to enrich our categories.

A *preorder-enriched category* is a category where each hom-set $Hom\ (A, B)$ is equipped with a preorder $\preceq$, requiring also that composition is monotonic with respect to these orderings:

$$f \preceq g \wedge h \preceq j \quad \Rightarrow \quad f \circ h \preceq g \circ j$$

Functors between preorder-enriched categories are also typically required to respect the ordering of arrows:

$$f \preceq g \quad \Rightarrow \quad \mathsf{F}\, f \preceq \mathsf{F}\, g$$

As arrows are used to model programs, the use of a preorder structure allows us to make ordering comparisons between programs. Any notion of improvement will lead to an ordering on programs, so we can use this machinery to make general arguments about improvement; while appeals to a particular semantics are needed to *establish* an ordering on programs, once such an ordering is in place we can continue reasoning with categorical techniques. Where before we used equational reasoning in our proofs, the preordering allows us to use the technique of *inequational* reasoning.

Any ordinary (locally small) category can be treated as a preorder-enriched category simply by equipping its hom-sets with the discrete ordering (i.e. $f \preceq f$ for all arrows f). Thus, any statement true of preorder-enriched categories can be specialised to a statement that is true of ordinary categories.

The idea of using preorder-enriched categories to compare programs was previously employed in the area of program *refinement* (Hoare and He, 1990; Back and Wright, 1998). While improvement is the problem of making a program more efficient, refinement is the related problem of making a program more *executable*, in the sense of transforming a specification into an implementation.

8.1.3 Other Work

Okasaki (1999) uses techniques of amortised cost analysis to reason about asymptotic time complexity of lazy functional data structures. This is achieved by modifying analysis techniques such as the Banker's Method, where the notion of *credit* is used to spread out the notional cost of an expensive but infrequent operations over more frequent and cheaper operations. The key idea in Okasaki's work is to invert such techniques to use the notion of *debt*. This allows the analyses to deal with the persistence of data structures, where the same structure may exist in multiple versions at once. While credit may only be spent once, a single debt may be paid off multiple times (in different versions of the same structure) without risking bankruptcy. These techniques have been used to analyse the asymptotic performance of a number of

functional data structures.

Sansom and Peyton Jones (1997) give a presentation of the GHC profiler, which can be used to measure time as well as space usage of Haskell programs. In doing so, they give a formal cost semantics for GHC Core programs based around the notion of *cost centres*. Cost centres are a way of annotating expressions so that the profiler can indicate which parts of the source program cost the most to execute. The cost semantics is used as a specification to develop a precise profiling framework, as well as to prove various properties about cost attribution and verify that certain program transformations do not affect the attribution of costs, though they may of course reduce cost overall. Cost centres are now one of the standard techniques for profiling Haskell programs.

Hope (2008) applies a technique based on *instrumenting* an abstract machine with cost information to derive cost semantics for call-by-value functional programs. More specifically, starting from a denotational semantics for the source language, one derives an abstract machine for this language using standard program transformation techniques, instruments this machine with cost information, and then reverses the derivation to arrive at an instrumented denotational semantics. This semantics can then be used to reason about the cost of programs in the high-level source language without reference to the details of the abstract machine. This approach was used to calculate the space and time cost of a range of programming examples, as well as to derive a new deforestation theorem for hylomorphisms.

Hammond and Michaelson (2003) present Hume, a domain-specific functional language for contexts where resources are constrained such as embedded and real-time systems. Hume aims to provide many of the programming features of a high-level functional language, while still making strong guarantees about space and time costs of programs. In order to facilitate programming for systems with limited memory, Hammond and Michaelson present a set of axioms that can be used to determine the space cost of evaluating an expression, allowing the space cost of programs to be

predicted at compile time.

A number of type-based approaches have also been investigated for reasoning about resource usage. Vasconcelos and Hammond (2003) present a system that can infer size and cost equations for recursive functional programs. This system is based on earlier work by Hughes, Pareto, and Sabry (1996) on *sized types*, type systems where recursive types are indexed by the recursion depth of their values. Furthermore, Hofmann and Jost (2003) present a type system for first-order functional programs that can be used to obtain linear bounds on heap space consumption. Of particular interest is work by Hofmann and Moser (2014) on *amortised resource analysis*, which combines type-based analysis with techniques from amortised cost analysis to give results that can be applied a wide range of different settings.

Chapter 9

Operational Improvement

"A LISP programmer knows the value of everything, but the cost of nothing."
— Alan Perlis, Epigrams on Programming

9.1 Motivation

Time and space usage are generally thought of as *operational* properties rather than denotational ones. These properties generally depend on the specifics of the algorithms used and the model of execution, details that most denotational semantics leave out. Therefore, the natural starting point for reasoning about efficiency is an operational theory. In this chapter, we prove efficiency properties about the worker/wrapper transformation using the operational *improvement theory*. Specifically, this chapter makes the following contributions:

- We show how work on *call-by-need* improvement theory by Moran and Sands (1999a) can be applied to formally justify that the worker/wrapper transformation for least fixed points preserves or improves time performance;
- We present preconditions that ensure the worker/wrapper transformation improves performance in this manner, which come naturally from the preconditions that ensure correctness;

- We demonstrate the utility of the new theory by verifying that examples from previous chapters indeed exhibit a time improvement.

The use of *call-by-need* improvement theory means that our work applies to lazy functional languages such as Haskell. Traditionally the operational beheaviour of lazy evaluation has been seen as difficult to reason about, but we show here that with the right tools this need not be the case.

Improvement theory does not seem to have attracted much attention in recent years, but we hope that this work can help to generate more interest in this and other techniques for reasoning about lazy evaluation.

9.2 Background: Improvement Theory

In order to develop a worker/wrapper theory that can prove *efficiency* properties, we need an operational theory of *program improvement*. More than just expressing extensional information, this should be based on intensional properties of resources that a program requires. For the purpose of this chapter, the resource we shall consider is execution time.

We have two main design goals for our operational theory. Firstly, it ought to be based on the operational semantics of a realistic programming language, so that conclusions we draw from it are as applicable as possible. Secondly, it should be amenable to techniques of (in)equational reasoning, as these are the techniques we used to develop the worker/wrapper correctness theory.

For the first goal, we use a language with similar syntax and semantics to GHC Core, except that arguments to functions are required to be atomic, as was the case in earlier versions of the core language (Peyton Jones, 1996). (Normalisation of the current version of GHC Core into this form is straightforward.) The language is call-by-need, reflecting the use of lazy evaluation in Haskell. The efficiency behaviour of call-by-need programs is notoriously counterintuitive. Our hope is that by providing

formal techniques for reasoning about call-by-need efficiency, we will go some way toward easing this problem.

For the second goal, our theory must be based around some relation R that is a preorder, as transitivity and reflexivity are both necessary for inequational reasoning to be valid. Furthermore, to support reasoning in a compositional manner, it is essential that our relation respect substitution. That is, given terms M and N, if $M\ R\ N$ then $\mathbb{C}[M]\ R\ \mathbb{C}[N]$ should also hold for any context $\mathbb{C}$. A relation R that satisfies both of these properties is called a *precongruence*.

A naïve approach to measuring execution time would be to simply count the number of steps taken to evaluate a term to some normal form, and consider that a term M is more efficient than a term N if its evaluation finishes in fewer steps. The resulting relation is clearly a preorder. However, it is not a precongruence in a call-by-need setting, because meaningful computations can be done with terms that are not fully normalised. Just because M normalises and N does not, it does not follow that M is necessarily always more efficient.

The approach we use is due to Moran and Sands (1999a). Rather than simply counting the steps taken to normalise a term in isolation, we compare the steps taken in *all contexts*, only saying that M is improved by N if for any context $\mathbb{C}$, the term $\mathbb{C}[M]$ requires no more evaluation steps than the term $\mathbb{C}[N]$. The result is a relation that is trivially a precongruence: it inherits transitivity and reflexivity from the numerical ordering $\leqslant$, and is substitutive by construction.

Improvement theory was originally developed in the context of call-by-name languages by Sands (1991). The remainder of this section reviews the call-by-need time improvement theory due to Moran and Sands (1999a), which will provide the setting for our operational worker/wrapper theory. The essential difference between call-by-name and call-by-need is that the latter implements a *sharing* strategy, avoiding the repeated evaluation of terms that are used more than once.

$$
\begin{array}{lll}
\langle \Gamma\ \{x = M\}, x, S\rangle & \to \langle \Gamma, M, \#x\ :\ S\rangle & \{\textsc{Lookup}\} \\
\langle \Gamma, V, \#x\ :\ S\rangle & \to \langle \Gamma\ \{x = V\}, V, S\rangle & \{\textsc{Update}\} \\
\langle \Gamma, M\ x, S\rangle & \to \langle \Gamma, M, x\ :\ S\rangle & \{\textsc{Unwind}\} \\
\langle \Gamma, \lambda x \to M, y\ :\ S\rangle & \to \langle \Gamma, M\ [y\ /\ x], S\rangle & \{\textsc{Subst}\} \\
\langle \Gamma, \mathbf{case}\ M\ \mathbf{of}\ alts, S\rangle & \to \langle \Gamma, M, alts\ :\ S\rangle & \{\textsc{Case}\} \\
\langle \Gamma, c_j\ \vec{y}, \{c_i\ \vec{x_i} \to N_i\}\ :\ S\rangle & \to \langle \Gamma, N_j\ [\vec{y}\ /\ \vec{x_j}], S\rangle & \{\textsc{Branch}\} \\
\langle \Gamma, \mathbf{let}\ \{\vec{x} = \vec{M}\}\ \mathbf{in}\ N, S\rangle & \to \langle \Gamma\ \{\vec{x} = \vec{M}\}, N, S\rangle & \{\textsc{Letrec}\}
\end{array}
$$

Figure 9.1: The call-by-need abstract machine

9.2.1 Operational Semantics of the Core Language

We shall begin by presenting the operational model that forms the basis of this improvement theory. The semantics presented here are based on Sestoft's Mark 1 abstract machine (Sestoft, 1997).

We start from a set of variables Var and a set of constructors Con. We assume all constructors have a fixed arity. The grammar of terms is as follows:

$$
\begin{array}{lll}
x, y, z & \in & Var \\
c & \in & Con \\
M, N & ::= & x \\
& | & \lambda x \to M \\
& | & M\ x \\
& | & \mathbf{let}\ \{\vec{x} = \vec{M}\}\ \mathbf{in}\ N \\
& | & c\ \vec{x} \\
& | & \mathbf{case}\ M\ \mathbf{of}\ \{c_i\ \vec{x_i} \to N_i\}
\end{array}
$$

We use $\vec{x} = \vec{M}$ as a shorthand for a list of bindings of the form $x = M$. Similarly, we use $c_i\ \vec{x_i} \to N_i$ as a shorthand for a list of cases of the form $c\ \vec{x} \to N$. All constructors are assumed to be saturated, that is, we assume that any $\vec{x}$ that is the operand of a constructor c has length equal to the arity of c. Literals are represented by constructors of arity 0. We treat α-equivalent terms as identical.

A term is a *value* if it is of the form $c\ \vec{x}$ or $\lambda x \to M$. In Haskell this is referred to as

a *weak head normal form*. We shall use letters such as V, W to denote value terms.

Term contexts take the following form, with substitution defined the obvious way.

$$
\begin{aligned}
\mathbb{C}, \mathbb{D} ::= {} & [-] \\
\mid {} & x \\
\mid {} & \lambda x \to \mathbb{C} \\
\mid {} & \mathbb{C}\ x \\
\mid {} & \mathbf{let}\ \{\vec{x} = \vec{\mathbb{C}}\}\ \mathbf{in}\ \mathbb{D} \\
\mid {} & c\ \vec{x} \\
\mid {} & \mathbf{case}\ \mathbb{C}\ \mathbf{of}\ \{c_i\ \vec{x_i} \to \mathbb{D}_i\}
\end{aligned}
$$

A value context is a context that is either a lambda abstraction or a constructor applied to variables.

The restriction that the arguments of functions and constructors always be variables has the effect that all bindings made during evaluation must have been created by a **let**. Sometimes we will use $M\ N$ (where N is not a variable) as a shorthand for $\mathbf{let}\ \{x = N\}\ \mathbf{in}\ M\ x$, where x is fresh. We use this shorthand for both terms and term contexts.

An abstract machine for executing terms in this language maintains as a state a triple $\langle \Gamma, M, S \rangle$ consisting of: the heap Γ, a set of bindings from variables to terms; the term M currently being evaluated; and the evaluation stack S, a list of tokens used by the abstract machine. The machine evaluates the current term to a value, and then decides what to do with the value based on the top of the stack. Bindings generated by **let** constructs are put on the heap, and only taken off when performing a LOOKUP. A LOOKUP executes by putting a token on the stack representing where the term was looked up, and then evaluating that term to value form before replacing it on the heap. In this way, each binding is only ever evaluated at most once. The semantics of the machine is given in Figure 9.1. Note that the LETREC rule assumes that $\vec{x}$ is disjoint from the domain of Γ; if not, we need only α-rename so that this is the case.

9.2.2 The Cost Model and Improvement Relations

Now that we have a semantics for our model, we must devise a *cost model* for this semantics. The natural way to do this for an operational semantics is to count steps taken to evaluate a given term. We use the notation $M{\downarrow^n}$ to mean the abstract machine progresses from the initial state $\langle \varnothing, M, \epsilon \rangle$ to some final state $\langle \Gamma, V, \epsilon \rangle$ with n occurrences of the LOOKUP step. It is sufficient to count LOOKUP steps because the total number of steps is bounded by a linear function of the number of LOOKUP steps (Moran and Sands, 1999a). Furthermore, we use the notation $M{\downarrow^{\leqslant n}}$ to mean that $M{\downarrow^m}$ for some $m \leqslant n$.

From this, we can define our improvement relation. We say that "M is *improved* by N", written $M \underset{\sim}{\rhd} N$, if the following statement holds for all contexts $\mathbb{C}$:

$$\mathbb{C}[M]{\downarrow^m} \implies \mathbb{C}[N]{\downarrow^{\leqslant m}}$$

In other words, a term M is improved by a term N if N takes no more steps to evaluate than M in all contexts. That this relation is a congruence follows immediately from the definition, and that it is a preorder follows from the fact that $\leqslant$ is itself a preorder. We sometimes write $M \underset{\sim}{\lhd} N$ for $N \underset{\sim}{\rhd} M$. If both $M \underset{\sim}{\rhd} N$ and $M \underset{\sim}{\lhd} N$, we write $M \underset{\sim}{\lhd\rhd} N$ and say that M and N are *cost-equivalent*.

For convenience, we define a "tick" operation on terms that adds exactly one unit of cost to a term:

$$\checkmark M \equiv \mathbf{let}\ \{x = M\}\ \mathbf{in}\ x \quad \{\text{where } x \text{ is free in } M\}$$

This definition for $\checkmark M$ takes exactly two steps to evaluate to M: one to add the binding to the heap, and the other to look it up. Only one of these steps is a LOOKUP step, so the result is that the cost of evaluating the term is increased by exactly one. Using ticks allows us to annotate terms with individual units of cost, allowing us to use rules to "push" cost around a term and thus make the calculations more convenient. We

could also define the tick operation by adding it to the grammar of terms and modifying the abstract machine and cost model accordingly, but this definition is equivalent. Clearly, ticks satisfy the following law:

$$\checkmark M \mathrel{\underset{\sim}{\triangleright}} M$$

The improvement relation $\mathrel{\underset{\sim}{\triangleright}}$ covers when one term is at least as efficient as another in all contexts, but this is a very strong statement. We use the notion of "weak improvement" when one term is at least as efficient as another within a constant factor. Specifically, we say M is weakly improved by N, written $M \mathrel{\underset{\approx}{\triangleright}} N$, if there exists a linear function $f(x) = kx + c$ (where $k, c \geqslant 0$) such that the following statement holds for all contexts $\mathbb{C}$:

$$\mathbb{C}[M]{\downarrow^m} \quad \Longrightarrow \quad \mathbb{C}[N]{\downarrow^{\leqslant f(m)}}$$

This can be read as "replacing M with N may make programs worse, but cannot make them *asymptotically* worse". We use symbols $\mathrel{\underset{\approx}{\triangleleft}}$ and $\mathrel{\underset{\approx}{\triangleleft\triangleright}}$ for inverse and equivalence analogously as for standard improvement.

Because weak improvement ignores constant factors, we have the following *tick introduction/elimination* law:

$$M \mathrel{\underset{\approx}{\triangleleft\triangleright}} \checkmark M$$

It follows from this that any improvement $M \mathrel{\underset{\sim}{\triangleright}} N$ can be *weakened* to a weak improvement $M' \mathrel{\underset{\approx}{\triangleright}} N'$ where M' and N' are the terms M and N with the ticks omitted.

The last notation we define is *entailment*, which is used when we have a chain of improvements that all apply with respect to a particular set of definitions. Specifically, where $\Gamma = \{\, \vec{x} = \vec{V} \,\}$ is a list of bindings, we write

$$\Gamma \vdash M_1 \mathrel{\underset{\sim}{\triangleright}} M_2 \mathrel{\underset{\sim}{\triangleright}} \cdots \mathrel{\underset{\sim}{\triangleright}} M_n$$

as a shorthand for

$$\mathbf{let}\ \Gamma\ \mathbf{in}\ M_1 \mathrel{\underset{\sim}{\triangleright}} \mathbf{let}\ \Gamma\ \mathbf{in}\ M_2 \mathrel{\underset{\sim}{\triangleright}} \cdots \mathrel{\underset{\sim}{\triangleright}} \mathbf{let}\ \Gamma\ \mathbf{in}\ M_n$$

9.2.3 Selected Laws

We finish this section with a selection of laws due to Moran and Sands (1999a). The first two are β-reduction rules. Firstly we have β-reduction for lambdas, which is a cost equivalence:

$$(\lambda x \to M)\ y \mathrel{\underset{\sim}{\lhd\rhd}} M\ [y\ /\ x]$$

This holds because the abstract machine evaluates the left-hand-side to the right-hand-side without performing any LOOKUPS, resulting in the same heap and stack as before. Note that the substitution is variable-for-variable, as the grammar for our language requires that the argument to function application always be a variable. In general, where a term M can be evaluated to a term M', we have the following:

$$M \mathrel{\underset{\sim}{\triangleright}} M'$$
$$M' \mathrel{\underset{\approx}{\lhd\rhd}} M$$

The latter fact may be non-obvious, but it holds because evaluating a term will produce a constant number of ticks, and tick-elimination is a weak cost-equivalence. In this manner we can see that partial evaluation by itself will never save more than a constant-factor of time.

The following cost equivalence allows us to substitute a variable for its binding. However, note that this is only valid for *values*, as bindings to other terms will be modified in the course of execution. We thus call this rule *value-*β.

$$\mathbf{let}\ \{x = V, \vec{y} = \vec{\mathbb{C}}[x]\}\ \mathbf{in}\ \mathbb{D}[x]$$
$$\mathrel{\underset{\sim}{\lhd\rhd}}$$
$$\mathbf{let}\ \{x = V, \vec{y} = \vec{\mathbb{C}}[\checkmark V]\}\ \mathbf{in}\ \mathbb{D}[\checkmark V]$$

The following law allows us to move bindings in and out of a context when the binding is to a value. Note that we assume that x does not appear in $\mathbb{C}$, which can be ensured by α-renaming, and that no free variables in V are captured in $\mathbb{C}$. We call this rule *value let-floating*.

$$\mathbb{C}[\mathbf{let}\ \{x = V\}\ \mathbf{in}\ M] \mathrel{\lhd\!\rhd}_{\sim} \mathbf{let}\ \{x = V\}\ \mathbf{in}\ \mathbb{C}[M]$$

We also have a *garbage collection* law allowing us to remove unused bindings. Assuming that x is not free in $\vec{N}$ or L, we have the following cost equivalence:

$$\mathbf{let}\ \{x = M; \vec{y} = \vec{N}\}\ \mathbf{in}\ L \mathrel{\lhd\!\rhd}_{\sim} \mathbf{let}\ \{\vec{y} = \vec{N}\}\ \mathbf{in}\ L$$

The final law we present here is the rule of *improvement induction*. The version that we present is stronger than the version in Moran and Sands (1999a), but can be obtained by a simple modification of the proof given there. For any set of value bindings Γ and context $\mathbb{C}$, we have the following rule:

$$\frac{\Gamma \vdash M \mathrel{\rhd}_{\sim} \checkmark\mathbb{C}[M] \qquad \Gamma \vdash \checkmark\mathbb{C}[N] \mathrel{\rhd}_{\sim} N}{\Gamma \vdash M \mathrel{\rhd}_{\sim} N}$$

This allows us to prove an $M \mathrel{\rhd}_{\sim} N$ simply by finding a context $\mathbb{C}$ where we can "unfold" M to $\checkmark\mathbb{C}[M]$ and "fold" $\checkmark\mathbb{C}[N]$ to N. In other words, the following apparently circular proof is valid:

$$\begin{array}{l} \Gamma \vdash M \\ \quad \mathrel{\rhd}_{\sim} \\ \quad\quad \checkmark\mathbb{C}[M] \\ \quad \mathrel{\rhd}_{\sim} \quad \{\,\text{hypothesis}\,\} \\ \quad\quad \checkmark\mathbb{C}[N] \\ \quad \mathrel{\rhd}_{\sim} \\ \quad\quad N \end{array}$$

This technique is similar to proof principles such as guarded coinduction (Coquand, 1993; Turner, 1995).

As a corollary to this law, we have the following law for *cost-equivalence* improvement induction. For any set of value bindings Γ and context $\mathbb{C}$, we have:

$$\frac{\Gamma \vdash M \mathrel{\underset{\sim}{\lhd\rhd}} \checkmark\mathbb{C}[M] \qquad \Gamma \vdash \checkmark\mathbb{C}[N] \mathrel{\underset{\sim}{\lhd\rhd}} N}{\Gamma \vdash M \mathrel{\underset{\sim}{\lhd\rhd}} N}$$

To prove this, we simply start from the assumptions and make two applications of improvement induction: first to prove $M \mathrel{\underset{\sim}{\rhd}} N$, and second to prove $N \mathrel{\underset{\sim}{\rhd}} M$.

9.3 Worker/Wrapper and Improvement

As we observed in Chapter 6, the essence of the worker/wrapper transformation is the application of one of the two related rules of *rolling* and *fusion*. It is natural to expect that a worker/wrapper transformation based in improvement theory will require its own versions of these rules. In this section, therefore, we first prove fusion and rolling rules for improvement theory, and then use these to prove a factorisation theorem for improvement theory analogous to the worker/wrapper factorisation theorem given in Chapter 4.

9.3.1 Preliminary Results

The first rule we prove corresponds to the *rolling rule*. In the context of improvement theory, this translates to the statement that for any pair of value contexts $\mathbb{F}$, $\mathbb{G}$ we have the following weak cost equivalence:

$$\mathbf{let}\ \{x = \mathbb{F}[\mathbb{G}[x]]\}\ \mathbf{in}\ \mathbb{G}[x] \mathrel{\underset{\approx}{\lhd\rhd}} \mathbf{let}\ \{x = \mathbb{G}[\mathbb{F}[x]]\}\ \mathbf{in}\ x$$

The proof begins with an application of cost-equivalence improvement induction. We let $\Gamma = \{x = \mathbb{F}[\checkmark\mathbb{G}[x]], y = \mathbb{G}[\checkmark\mathbb{F}[y]]\}$, $M = \checkmark\mathbb{G}[x]$, $N = y$, $\mathbb{C} = \mathbb{G}[\checkmark\mathbb{F}[-]]$. The premises of induction are proved as follows:

$\Gamma \vdash M$

$$
\begin{array}{ll}
\equiv & \{\text{ definitions }\} \\
& \checkmark\mathbb{G}[x] \\
\lhd\!\rhd & \{\text{ value-}\beta\text{ }\} \\
& \checkmark\mathbb{G}[\checkmark\mathbb{F}[\checkmark\mathbb{G}[x]]] \\
\equiv & \{\text{ definitions }\} \\
& \checkmark\mathbb{C}[M]
\end{array}
$$

and

$$
\begin{array}{ll}
\Gamma \vdash \checkmark\mathbb{C}[N] & \\
\equiv & \{\text{ definitions }\} \\
& \checkmark\mathbb{G}[\checkmark\mathbb{F}[y]] \\
\lhd\!\rhd & \{\text{ value-}\beta\text{ }\} \\
& y \\
\equiv & \{\text{ definitions }\} \\
& N
\end{array}
$$

Thus we can conclude $\Gamma \vdash M \lhd\!\rhd N$, or equivalently $\mathbf{let}\ \Gamma\ \mathbf{in}\ M \lhd\!\rhd \mathbf{let}\ \Gamma\ \mathbf{in}\ N$. We expand this out and apply garbage collection to remove the unused bindings:

$$\mathbf{let}\ \{x = \mathbb{F}[\checkmark\mathbb{G}[x]]\}\ \mathbf{in} \checkmark\mathbb{G}[x] \lhd\!\rhd \mathbf{let}\ \{y = \mathbb{G}[\checkmark\mathbb{F}[y]]\}\ \mathbf{in}\ y$$

By applying α-renaming and weakening we obtain the desired result

The second rule we prove is *letrec-fusion*, analogous to fixed-point fusion. For any value contexts $\mathbb{F}, \mathbb{G}$, we have the following implication:

$$
\begin{array}{l}
\mathbb{H}[\checkmark\mathbb{F}[x]] \rhd \mathbb{G}[\checkmark\mathbb{H}[x]] \\
\Rightarrow \\
\mathbf{let}\ \{x = \mathbb{F}[x]\}\ \mathbf{in}\ \mathbb{H}[x] \rhd \mathbf{let}\ \{x = \mathbb{G}[x]\}\ \mathbf{in}\ x
\end{array}
$$

For the proof, we assume the premise and proceed by improvement induction. Let $\Gamma = \{x = \mathbb{F}[x], y = \mathbb{G}[y]\}, M = \checkmark\mathbb{H}[x], N = y, \mathbb{C} = \mathbb{G}$. The premises are proved as follows:

$$
\begin{array}{ll}
\Gamma \vdash M & \\
\quad \equiv & \{ \text{by definitions} \} \\
\quad\quad \checkmark\mathbb{H}[x] & \\
\quad \mathrel{\underset{\sim}{\lhd\rhd}} & \{ \text{value beta} \} \\
\quad\quad \checkmark\mathbb{H}[\checkmark\mathbb{F}[x]] & \\
\quad \mathrel{\underset{\sim}{\rhd}} & \{ \text{by assumption} \} \\
\quad\quad \checkmark\mathbb{G}[\checkmark\mathbb{H}[x]] & \\
\quad \equiv & \{ \text{definition} \} \\
\quad\quad \checkmark\mathbb{C}[M] &
\end{array}
$$

and

$$
\begin{array}{ll}
\Gamma \vdash \checkmark\mathbb{C}[N] & \\
\quad \equiv & \{ \text{by definitions} \} \\
\quad\quad \checkmark\mathbb{G}[y] & \\
\quad \mathrel{\underset{\sim}{\lhd\rhd}} & \{ \text{value beta} \} \\
\quad\quad y & \\
\quad \equiv & \{ \text{definition} \} \\
\quad\quad N &
\end{array}
$$

Thus we conclude that $\Gamma \vdash M \mathrel{\underset{\sim}{\rhd}} N$. Expanding and applying garbage collection, we obtain the following:

$$\mathbf{let}\ \{x = \mathbb{F}[x]\}\ \mathbf{in} \checkmark\mathbb{H}[x] \mathrel{\underset{\sim}{\rhd}} \mathbf{let}\ y = \mathbb{G}[y]\ \mathbf{in}\ y$$

Again we obtain the desired result via weakening and α-renaming. As improvement induction is symmetrical, we can also prove the following dual fusion law, in which the improvement relations are reversed:

$$
\begin{array}{l}
\quad \mathbb{H}[\checkmark\mathbb{F}[x]] \mathrel{\underset{\sim}{\lhd}} \mathbb{G}[\checkmark\mathbb{H}[x]] \\
\Rightarrow \\
\quad \mathbf{let}\ \{x = \mathbb{F}[x]\}\ \mathbf{in}\ \mathbb{H}[x] \mathrel{\underset{\approx}{\lhd}} \mathbf{let}\ \{x = \mathbb{G}[x]\}\ \mathbf{in}\ x
\end{array}
$$

For both the rolling and fusion rules, we first proved a version of the conclusion with normal improvement, and then weakened to weak improvement. We do this to avoid having to deal with ticks, and because the weaker version is strong enough for our purposes.

Our fusion law differs from the fusion law that was originally presented by Moran and Sands. Their law requires that the context $\mathbb{H}$ satisfy a form of *strictness*. Specifically, for any value contexts $\mathbb{F}, \mathbb{G}$ and fresh variable x, the following implication holds:

$$\begin{aligned} & \mathbb{H}[\mathbb{F}[x]] \trianglerighteq \mathbb{G}[\mathbb{H}[x]] \wedge \mathit{strict}\ (\mathbb{H}) \\ \Rightarrow & \\ & \mathbf{let}\ \{x = \mathbb{F}[x]\}\ \mathbf{in}\ \mathbb{C}[\mathbb{H}[x]] \trianglerighteq \mathbf{let}\ \{x = \mathbb{G}[x]\}\ \mathbf{in}\ \mathbb{C}[x] \end{aligned}$$

This version of fusion has the advantage of having a stronger conclusion, but its strictness side-condition and lack of symmetry make it unsuitable for our purposes.

9.3.2 The Worker/Wrapper Improvement Theorem

Using the above set of rules, we can prove the *worker/wrapper improvement* theorem in Figure 9.2, giving conditions under which a program factorisation is a time improvement. Given a recursive program $\mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ x$ and *abstraction* and *representation* contexts Abs and Rep, this theorem gives us conditions we can use to derive a factorised program $\mathbf{let}\ x = \mathbb{G}[x]\ \mathbf{in}\ \mathsf{Abs}[x]$. This factorised program will be at worst a constant factor slower than the original program, but can potentially be asymptotically faster. In other words, we have conditions that guarantee that such an optimisation is "safe" with respect to time performance.

The proofs for the correctness theorems center on the use of the rolling and fusion rules. Because we have proven analogous rules in our setting, the proofs can be adapted fairly straightforwardly, simply by keeping the general form of the proofs and using the rules of improvement theory as structural rules that fit between the original steps. The details are as follows.

Given value contexts Abs, Rep, $\mathbb{F}$, $\mathbb{G}$ for which x is free satisfying one of the assumptions

$$\begin{array}{ll} \text{(A)} & \mathsf{Abs}[\mathsf{Rep}[x]] \mathrel{\underset{\approx}{\lhd\rhd}} x \\ \text{(B)} & \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[x]]] \mathrel{\underset{\approx}{\lhd\rhd}} \mathbb{F}[x] \\ \text{(C)} & \mathbf{let}\ x = \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[x]]]\ \mathbf{in}\ x \mathrel{\underset{\approx}{\lhd\rhd}} \mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ x \end{array}$$

If any one of the following conditions holds:

$$\begin{array}{ll} \text{(1)} & \mathbb{G}[x] \mathrel{\underset{\approx}{\lhd}} \mathsf{Rep}[\mathbb{F}[\mathsf{Abs}[x]]] \\ \text{(2)} & \mathbb{G}[\checkmark\mathsf{Rep}[x]] \mathrel{\underset{\sim}{\lhd}} \mathsf{Rep}[\checkmark\mathbb{F}[x]] \\ \text{(3)} & \mathsf{Abs}[\checkmark\mathbb{G}[x]] \mathrel{\underset{\sim}{\lhd}} \mathbb{F}[\checkmark\mathsf{Abs}[x]] \end{array}$$

$$\begin{array}{ll} \text{(1}\beta\text{)} & \mathbf{let}\ x = \mathbb{G}[x]\ \mathbf{in}\ x \mathrel{\underset{\approx}{\lhd}} \mathbf{let}\ x = \mathsf{Rep}[\mathbb{F}[\mathsf{Abs}[x]]]\ \mathbf{in}\ x \\ \text{(2}\beta\text{)} & \mathbf{let}\ x = \mathbb{G}[x]\ \mathbf{in}\ x \mathrel{\underset{\approx}{\lhd}} \mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ \mathsf{Rep}[x] \end{array}$$

then we have the improvement:

$$\mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ x \mathrel{\underset{\approx}{\rhd}} \mathbf{let}\ x = \mathbb{G}[x]\ \mathbf{in}\ \mathsf{Abs}[x]$$

Figure 9.2: The Worker/Wrapper Improvement Theorem

We begin by noting that (A) $\Rightarrow$ (B) $\Rightarrow$ (C), as in the original case. The first implication (A) $\Rightarrow$ (B) no longer follows immediately, but the proof is simple. Letting y be a fresh variable, we reason as follows:

$$\begin{array}{ll} & \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[y]]] \\ \mathrel{\underset{\approx}{\lhd\rhd}} & \{\ \text{garbage collection, value-}\beta\ \} \\ & \mathbf{let}\ x = \mathbb{F}[y]\ \mathbf{in}\ \mathsf{Abs}[\mathsf{Rep}[x]] \\ \mathrel{\underset{\approx}{\lhd\rhd}} & \{\ \text{(A)}\ \} \\ & \mathbf{let}\ x = \mathbb{F}[y]\ \mathbf{in}\ x \\ \mathrel{\underset{\approx}{\lhd\rhd}} & \{\ \text{value-}\beta\text{, garbage collection}\ \} \\ & \mathbb{F}[y] \end{array}$$

The final step is to observe that as both x and y are fresh, we can substitute one for the other and the relationship between the terms will remain the same. Hence, we can conclude (B).

As in the original theorem, we have that (1) implies (1β) by simple application of substitution, (2) implies (2β) by fusion and (3) implies the conclusion also by fusion. Under assumption (C), we have that (1β) and (2β) are equivalent. We show this by proving their right hand sides cost-equivalent, after which we can apply transitivity.

$$
\begin{array}{ll}
& \mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ \mathsf{Rep}[x] \\
\lhd\!\rhd_{\approx} & \{\ \text{value-}\beta\ \} \\
& \mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ \mathsf{Rep}[\mathbb{F}[x]] \\
\lhd\!\rhd_{\approx} & \{\ \text{value let-floating}\ \} \\
& \mathsf{Rep}[\mathbb{F}[\mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ x]] \\
\lhd\!\rhd_{\approx} & \{\ (\mathsf{C})\ \} \\
& \mathsf{Rep}[\mathbb{F}[\mathbf{let}\ x = \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[x]]]\ \mathbf{in}\ x]] \\
\lhd\!\rhd_{\approx} & \{\ \text{value let-floating}\ \} \\
& \mathbf{let}\ x = \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[x]]]\ \mathbf{in}\ \mathsf{Rep}[\mathbb{F}[x]] \\
\lhd\!\rhd_{\approx} & \{\ \text{rolling}\ \} \\
& \mathbf{let}\ x = \mathsf{Rep}[\mathbb{F}[\mathsf{Abs}[x]]]\ \mathbf{in}\ x
\end{array}
$$

Finally, we must show that condition (1β) and assumption (C) together imply the conclusion. This follows exactly the same pattern of reasoning as the original proofs, with the addition of two applications of value-let floating:

$$
\begin{array}{ll}
& \mathbf{let}\ x = \mathbb{F}[x]\ \mathbf{in}\ x \\
\lhd\!\rhd_{\approx} & \{\ (\mathsf{C})\ \} \\
& \mathbf{let}\ x = \mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[x]]]\ \mathbf{in}\ x \\
\lhd\!\rhd_{\approx} & \{\ \text{rolling}\ \} \\
& \mathbf{let}\ x = \mathsf{Rep}[\mathbb{F}[\mathsf{Abs}[x]]]\ \mathbf{in}\ \mathsf{Abs}[x] \\
\lhd\!\rhd_{\approx} & \{\ \text{value let-floating}\ \} \\
& \mathsf{Abs}[\mathbf{let}\ x = \mathsf{Rep}[\mathbb{F}[\mathsf{Abs}[x]]]\ \mathbf{in}\ x] \\
\rhd_{\approx} & \{\ (1\beta)\ \} \\
& \mathsf{Abs}[\mathbf{let}\ x = \mathbb{G}[x]\ \mathbf{in}\ x]
\end{array}
$$

$\lhd\rhd_{\approx}$ { value let-floating }

$\mathbf{let}\ x = \mathbb{G}[x]\ \mathbf{in}\ \mathsf{Abs}[x]$

We conclude this section by discussing a few important points about the worker/wrapper improvement theorem and its applications. Firstly, we note that the condition (A) will never actually hold. To see this, we let Ω be a divergent term; that is, one that the abstract machine will never finish evaluating. By substituting into the context $\mathbf{let}\ x = \Omega\ \mathbf{in}\ [-]$, we obtain the following cost-equivalence:

$$\mathbf{let}\ x = \Omega\ \mathbf{in}\ \mathsf{Abs}[\mathsf{Rep}[x]] \lhd\rhd_{\approx} \mathbf{let}\ x = \Omega\ \mathbf{in}\ x$$

This is clearly false, as the left-hand side will terminate almost immediately (as Abs is a value context), while the right-hand side will diverge. Thus we see that assumption (A) is impossible to satisfy. We leave it in the theorem for completeness of the analogy with the earlier worker/wrapper theorems. In situations where (A) would have been used with the earlier theory, the weaker assumption (B) can always be used instead. As we will see later with the examples, frequently only very few properties of the context $\mathbb{F}$ will be used in the proof of (B). A *typed* improvement theory might allow these properties to be assumed of x instead, thus making (A) usable again.

Secondly, we note the restriction to value contexts. This is not actually a particularly severe restriction: for the common application of recursively-defined functions, it is fairly straightforward to ensure that all contexts be of the form $\lambda x\ \rightarrow\ \mathbb{C}$. For other applications it may be more difficult to find Abs and Rep contexts that satisfy the required relationship.

Finally, we note that only conditions (2) and (3) use normal improvement, with all other assumptions and conditions using the weaker version. This is because weak improvement is not strong enough to permit the use of fusion, which these conditions rely on. This makes these conditions harder to prove. However, when these conditions are used, their strength allows us to narrow down the source of any constant-factor slowdown that may take place.

9.4 Examples

9.4.1 Reversing a List

In this section we shall demonstrate the utility of our theory with two practical examples. We begin by revisiting the earlier example of reversing a list. In order to apply our theory, we must first write *reverse* as a recursive let:

$$
\begin{aligned}
\mathit{reverse} \quad &= \mathbf{let}\ \{f = \mathsf{Revbody}\ [f]\}\ \mathbf{in}\ f \\
\mathsf{Revbody}[M] &= \lambda xs \to \mathbf{case}\ xs\ \mathbf{of} \\
&\qquad\qquad [\,] \to [\,] \\
&\qquad\qquad (y : ys) \to M\ ys +\!\!+ [y]
\end{aligned}
$$

The *abs* and *rep* functions from before give rise to to the following contexts:

$$
\begin{aligned}
\mathsf{Abs}[M] &= \lambda xs \to M\ xs\ [\,] \\
\mathsf{Rep}[M] &= \lambda xs \to \lambda ys \to M\ xs +\!\!+ ys
\end{aligned}
$$

We also require some extra theoretical machinery that we have yet to introduce. To start with, we must assume some rules about the append operation $+\!\!+$. The following associativity rules were proved by Moran and Sands (1999a).

$$
\begin{aligned}
(xs +\!\!+ ys) +\!\!+ zs &\trianglerighteq xs +\!\!+ (ys +\!\!+ zs) \\
xs +\!\!+ (ys +\!\!+ zs) &\trianglelefteq (xs +\!\!+ ys) +\!\!+ zs
\end{aligned}
$$

We assume the following identity improvement as well, which follows from theorems also proved by Moran and Sands (1999a):

$$
[\,] +\!\!+ xs \trianglerighteq xs
$$

We also require the notion of an *evaluation context*. An evaluation context is a context where evaluation is impossible unless the hole is filled. Such contexts are of the following form:

$$
\begin{aligned}
\mathbb{E} ::= {} & \mathbb{A} \\
& \mid \mathbf{let}\ \{\vec{x} = \vec{M}\}\ \mathbf{in}\ \mathbb{A} \\
& \mid \mathbf{let}\ \{\vec{y} = \vec{M}; \\
& \qquad x_0 = \mathbb{A}_0[x_1]; \\
& \qquad x_1 = \mathbb{A}_1[x_2]; \\
& \qquad \ldots \\
& \qquad x_n = \mathbb{A}_n\} \\
& \quad \mathbf{in}\ \mathbb{A}[x_0]
\end{aligned}
$$

$$
\begin{aligned}
\mathbb{A} ::= {} & [-] \\
\mid {} & \mathbb{A}\ x \\
\mid {} & \mathbf{case}\ \mathbb{A}\ \mathbf{of}\ \{c_i\ \vec{x_i} \to M_i\}
\end{aligned}
$$

Note that a context of this form must have exactly one hole. The usefulness of evaluation contexts is that they satisfy some special laws. We use the following laws in this particular example:

$$
\begin{aligned}
& \mathbb{E}[\checkmark M] \\
\lhd\!\rhd_{\sim} \quad & \{\text{tick floating}\} \\
& \checkmark \mathbb{E}[M]
\end{aligned}
$$

$$
\begin{aligned}
& \mathbb{E}[\mathbf{case}\ M\ \mathbf{of}\ \{c_i\ \vec{x_i} \to N_i\}] \\
\lhd\!\rhd_{\sim} \quad & \{\text{case floating}\} \\
& \mathbf{case}\ M\ \mathbf{of}\ \{c_i\ \vec{x_i} \to \mathbb{E}[N_i]\}
\end{aligned}
$$

$$
\begin{aligned}
& \mathbb{E}[\mathbf{let}\ \{\vec{x} = \vec{M}\}\ \mathbf{in}\ N] \\
\lhd\!\rhd_{\sim} \quad & \{\text{let floating}\} \\
& \mathbf{let}\ \{\vec{x} = \vec{M}\}\ \mathbf{in}\ \mathbb{E}[N]
\end{aligned}
$$

We conclude by noting that while the context $[-] +\!\!+ ys$ is not strictly speaking an evaluation context (as the hole is in the wrong place), it is cost-equivalent to an evaluation context and so also satisfies these laws. The proof is as follows:

$$
\begin{array}{ll}
& [-] +\!\!+ ys \\
\equiv & \quad \{ \text{desugaring} \} \\
& (\mathbf{let}\ \{xs = [-]\}\ \mathbf{in}\ (+\!\!+)\ xs)\ ys \\
\lhd\!\rhd & \quad \{ \text{let floating}\ [-]\ ys \} \\
& \mathbf{let}\ \{xs = [-]\}\ \mathbf{in}\ (+\!\!+)\ xs\ ys \\
\lhd\!\rhd & \quad \{ \text{unfolding} +\!\!+ \} \\
& \mathbf{let}\ \{xs = [-]\}\ \mathbf{in} \\
& \quad \checkmark \mathbf{case}\ xs\ \mathbf{of} \\
& \qquad [\,] \to ys \\
& \qquad (z : zs) \to \mathbf{let}\ \{rs = (+\!\!+)\ zs\ ys\}\ \mathbf{in}\ z : rs \\
\lhd\!\rhd & \quad \{ \text{desugaring tick and collecting lets} \} \\
& \mathbf{let}\ \{ \quad xs = [-]; \\
& \qquad r = \mathbf{case}\ xs\ \mathbf{of} \\
& \qquad\qquad [\,] \to ys \\
& \qquad\qquad (z : zs) \to \mathbf{let}\ \{rs = (+\!\!+)\ zs\ ys\}\ \mathbf{in}\ z : rs \\
& \quad \}\ \mathbf{in}\ r
\end{array}
$$

Now we can begin the example proper. We start by verifying that Abs and Rep satisfy one of the worker/wrapper assumptions. While earlier we used (A) for this example, the corresponding assumption for worker/wrapper improvement is unsatisfiable. Thus we instead verify assumption (B). The proof is fairly straightforward:

$$
\begin{array}{ll}
& \mathsf{Abs}[\mathsf{Rep}[\mathsf{Revbody}[f]]] \\
\equiv & \quad \{ \text{definitions} \} \\
& \lambda xs \to (\lambda xs \to \lambda ys \to \mathsf{Revbody}[f]\ xs +\!\!+ ys)\ xs\ [\,] \\
\lhd\!\rhd & \quad \{ \beta\text{-reduction} \} \\
& \lambda xs \to \mathsf{Revbody}[f]\ xs +\!\!+ [\,] \\
\equiv & \quad \{ \text{definition of Revbody} \} \\
& \lambda xs \to (\lambda xs \to \mathbf{case}\ xs\ \mathbf{of}
\end{array}
$$

$$
\begin{array}{ll}
 & \quad [\,] \to [\,] \\
 & \quad (y : ys) \to f\ ys +\!\!+ [y])\ xs +\!\!+ [\,] \\
\lhd\!\!\rhd & \quad \{\,\beta\text{-reduction}\,\} \\
 & \lambda xs \to (\mathbf{case}\ xs\ \mathbf{of} \\
 & \quad [\,] \to [\,] \\
 & \quad (y : ys) \to f\ ys +\!\!+ [y]) +\!\!+ [\,] \\
\lhd\!\!\rhd & \quad \{\,\text{case floating } [-] +\!\!+ [\,]\,\} \\
 & \lambda xs \to \mathbf{case}\ xs\ \mathbf{of} \\
 & \quad [\,] \to [\,] +\!\!+ [\,] \\
 & \quad (y : ys) \to (f\ ys +\!\!+ [y]) +\!\!+ [\,] \\
\lhd\!\!\rhd & \quad \{\,\text{associativity is weak cost equivalence}\,\} \\
 & \lambda xs \to \mathbf{case}\ xs\ \mathbf{of} \\
 & \quad [\,] \to [\,] +\!\!+ [\,] \\
 & \quad (y : ys) \to f\ ys +\!\!+ ([y] +\!\!+ [\,]) \\
\lhd\!\!\rhd & \quad \{\,\text{evaluating } [\,] +\!\!+ [\,], [y] +\!\!+ [\,]\,\} \\
 & \lambda xs \to \mathbf{case}\ xs\ \mathbf{of} \\
 & \quad [\,] \to [\,] \\
 & \quad (y : ys) \to f\ ys +\!\!+ [y] \\
\equiv & \quad \{\,\text{definition of revbody}\,\} \\
 & \text{Revbody}\ [f\,]
\end{array}
$$

As before, we use condition (2) to derive our $\mathbb{G}$. The derivation is somewhat more involved than before, requiring some care with the manipulation of ticks.

$$
\begin{array}{ll}
 & \text{Rep}[\,\checkmark \text{Revbody}[f\,]\,] \\
\equiv & \quad \{\,\text{definitions}\,\} \\
 & \lambda xs \to \lambda ys \to \\
 & \quad (\checkmark \lambda xs \to \quad \mathbf{case}\ xs\ \mathbf{of} \\
 & \qquad\qquad\qquad [\,] \to [\,]
\end{array}
$$

$$(z : zs) \to f\ zs +\!\!+ [z])\ xs +\!\!+ ys$$

$\lhd\!\rhd$ { float tick out of $[-]\ xs +\!\!+ ys$ }

$$\lambda xs \to \lambda ys \to$$
$$\checkmark((\lambda xs \to \ \mathbf{case}\ xs\ \mathbf{of}$$
$$[\,] \to [\,]$$
$$(z : zs) \to f\ zs +\!\!+ [z])\ xs +\!\!+ ys)$$

$\lhd\!\rhd$ { β-reduction }

$$\lambda xs \to \lambda ys \to \checkmark((\mathbf{case}\ xs\ \mathbf{of}$$
$$[\,] \to [\,]$$
$$(z : zs) \to f\ zs +\!\!+ [z]) +\!\!+ ys)$$

$\lhd\!\rhd$ { case floating $[-] +\!\!+ ys$ }

$$\lambda xs \to \lambda ys \to \checkmark(\mathbf{case}\ xs\ \mathbf{of}$$
$$[\,] \to [\,] +\!\!+ ys$$
$$(z : zs) \to (f\ zs +\!\!+ [z]) +\!\!+ ys)$$

$\rhd$ { associativity and identity of $+\!\!+$ }

$$\lambda xs \to \lambda ys \to \checkmark(\mathbf{case}\ xs\ \mathbf{of}$$
$$[\,] \to ys$$
$$(z : zs) \to f\ zs +\!\!+ ([z] +\!\!+ ys))$$

$\rhd$ { evaluating $[y] +\!\!+ ys$ }

$$\lambda xs \to \lambda ys \to \checkmark(\mathbf{case}\ xs\ \mathbf{of}$$
$$[\,] \to ys$$
$$(z : zs) \to f\ zs +\!\!+ (z : ys))$$

$\lhd\!\rhd$ { case floating tick $(\star)$ }

$$\lambda xs \to \lambda ys \to \mathbf{case}\ xs\ \mathbf{of}$$
$$[\,] \to \checkmark ys$$
$$(z : zs) \to \checkmark(f\ zs +\!\!+ (z : ys))$$

$\rhd$ { removing a tick }

$$\lambda xs \to \lambda ys \to \mathbf{case}\ xs\ \mathbf{of}$$

$$
\begin{array}{ll}
 & \quad [\,] \to ys \\
 & \quad (z : zs) \to \checkmark (f\ zs +\!\!+ (z : ys)) \\
\lhd\!\rhd & \quad \{\ \text{desugaring}\ \} \\
 & \lambda xs \to \lambda ys \to \mathbf{case}\ xs\ \mathbf{of} \\
 & \quad [\,] \to ys \\
 & \quad (z : zs) \to \checkmark (\mathbf{let}\ ws = (z : ys)\ \mathbf{in}\ f\ zs +\!\!+ ws) \\
\lhd\!\rhd & \quad \{\ \beta\text{-expansion}\ \} \\
 & \lambda xs \to \lambda ys \to \mathbf{case}\ xs\ \mathbf{of} \\
 & \quad [\,] \to ys \\
 & \quad (z : zs) \to \checkmark \mathbf{let}\ ws = (z : ys)\ \mathbf{in} \\
 & \qquad\qquad (\lambda as \to \lambda bs \to f\ as +\!\!+ bs)\ zs\ ws \\
\lhd\!\rhd & \quad \{\ \text{tick floating}\ [-]\ zs\ ws\ \} \\
 & \lambda xs \to \lambda ys \to \mathbf{case}\ xs\ \mathbf{of} \\
 & \quad [\,] \to ys \\
 & \quad (z : zs) \to \mathbf{let}\ ws = (z : ys)\ \mathbf{in} \\
 & \qquad\qquad (\checkmark \lambda as \to \lambda bs \to f\ as +\!\!+ bs)\ zs\ ws \\
\equiv & \quad \{\ \text{definition of Rep}\ \} \\
 & \lambda xs \to \lambda ys \to \mathbf{case}\ xs\ \mathbf{of} \\
 & \quad [\,] \to ys \\
 & \quad (z : zs) \to \mathbf{let}\ ws = (z : ys)\ \mathbf{in}\ (\checkmark \mathsf{Rep}[f])\ zs\ ws \\
\equiv & \quad \{\ \text{taking this as our definition of } \mathbb{G}\ \} \\
 & \mathbb{G}[\checkmark \mathsf{Rep}[f]]
\end{array}
$$

The step marked $\star$ is valid because $\checkmark[-]$ is itself an evaluation context, being syntactic sugar for $\mathbf{let}\ x = [-]\ \mathbf{in}\ x$. Thus we have derived a definition of $\mathbb{G}$, from which we create the following factorised program:

$$
\begin{array}{l}
reverse = \mathbf{let}\ \{rec = \mathbb{G}[rec]\}\ \mathbf{in}\ \mathsf{Abs}[rec] \\
\mathbb{G}[rec] = \lambda xs \to \lambda ys \to \mathbf{case}\ xs\ \mathbf{of}
\end{array}
$$

$$
\begin{array}{l}
\quad [\,] \to ys \\
\quad (z : zs) \to \mathbf{let}\ ws = (z : ys)\ \mathbf{in}\ rec\ zs\ ws
\end{array}
$$

Expanding this out, we obtain:

$$
\begin{array}{l}
reverse = \mathbf{let}\ \ \{ \\
\qquad rec = \lambda xs \to \lambda ys \to \mathbf{case}\ xs\ \mathbf{of} \\
\qquad\qquad\qquad\qquad [\,] \to ys \\
\qquad\qquad\qquad\qquad (z : zs) \to \mathbf{let}\ ws = (z : ys) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ rec\ zs\ ws \\
\qquad \} \\
\mathbf{in} \quad \lambda xs \to rec\ xs\ [\,]
\end{array}
$$

The result is an implementation of fast reverse as a recursive let. The calculations here have essentially the same structure as the correctness proofs, with the addition of some administrative steps to do with the manipulation of ticks.

9.4.2 Tabulating a Function

Our second example is that of tabulating a function by producing a stream (infinite list) of results. Recall that such a function can be implemented in Haskell as follows:

$$
\begin{array}{l}
tabulate :: (Int \to a) \to Stream\ a \\
tabulate\ f = f\ 0 : tabulate\ (f \circ (+1))
\end{array}
$$

As noted before, this definition repeats a great deal of work, as each result of the output stream is the result of evaluating a term of the form $(f \circ (+1) \circ (+1) \circ \ldots \circ (+1))\ 0$. We wish to apply the worker/wrapper technique to improve the time performance of this program. The first step is to write it as a recursive let in our language:

$$
tabulate = \mathbf{let}\ \{h = \mathbb{F}[h]\}\ \mathbf{in}\ h
$$

$$\begin{aligned}\mathbb{F}[M] \quad &= \lambda f \to \mathbf{let}\ \{f' = \lambda x \to \mathbf{let}\ \{x' = x + 1\}\ \mathbf{in}\ f\ x'\} \\ &\qquad \mathbf{in}\ f\ 0\ :\ M\ f'\end{aligned}$$

Next, we must devise Abs and Rep contexts. In order to avoid the repeated work, we hope to derive a version of the *tabulate* function that takes an additional number argument telling it where to "start" from. The following Abs and Rep contexts convert between these two versions:

$$\mathsf{Abs}[M] = \lambda f \to M\ 0\ f$$

$$\begin{aligned}\mathsf{Rep}[M] = \lambda n \to \lambda f \to &\mathbf{let}\ \{f' = \lambda x \to \mathbf{let}\ \{x' = x + n\}\ \mathbf{in}\ f\ x'\} \\ &\mathbf{in}\ M\ f'\end{aligned}$$

Once again, we must introduce some new rules before we can derive the factorised program. Firstly, we require the following two *variable substitution* rules from Moran and Sands (1999a):

$$\mathbf{let}\ \{x = y\}\ \mathbf{in}\ \mathbb{C}\ [x] \underset{\approx}{\rhd} \mathbf{let}\ \{x = y\}\ \mathbf{in}\ \mathbb{C}\ [y]$$

$$\mathbf{let}\ \{x = y\}\ \mathbf{in}\ \mathbb{C}\ [y] \underset{\approx}{\lhd\rhd} \mathbf{let}\ \{x = y\}\ \mathbf{in}\ \mathbb{C}\ [x]$$

Next, we must use some properties of addition. Firstly, we have the following *identity* properties:

$$x + 0 \underset{\sim}{\lhd\rhd} x$$

$$0 + x \underset{\sim}{\lhd\rhd} x$$

We also use the following property, combining associativity and commutativity. We shall refer to this as *associativity of* $+$. Where t is not free in $\mathbb{C}$, we have:

$$\begin{aligned}&\mathbf{let}\ \{t = x + y\}\ \mathbf{in} \\ &\quad \mathbf{let}\ \{r = t + z\}\ \mathbf{in}\ \mathbb{C}\ [r] \\ \underset{\sim}{\lhd\rhd} \\ &\mathbf{let}\ \{t = z + y\}\ \mathbf{in} \\ &\quad \mathbf{let}\ \{r = x + t\}\ \mathbf{in}\ \mathbb{C}\ [r]\end{aligned}$$

Finally, we use the fact that sums may be floated out of arbitrary contexts. Where z does not occur in $\mathbb{C}$, we have:

$$\mathbb{C}\,[\mathbf{let}\,\{z = y + x\}\,\mathbf{in}\,M] \mathrel{\underset{\sim}{\lhd\rhd}} \mathbf{let}\,\{z = y + x\}\,\mathbf{in}\,\mathbb{C}\,[M]$$

Now we can begin to apply worker/wrapper. Firstly, we verify that Abs and Rep satisfy assumption (B). Again, this is relatively straightforward:

$$\begin{aligned}
&\mathsf{Abs}[\mathsf{Rep}[\mathbb{F}[h]]] \\
\equiv\quad &\{\,\text{definitions}\,\} \\
&\lambda f \to (\lambda n \to \lambda f \to\ \ \mathbf{let}\,\{f' = \lambda x \to \mathbf{let}\,\{x' = x + n\}\,\mathbf{in}\,f\,x'\} \\
&\qquad\qquad\qquad\qquad \mathbf{in}\,\mathbb{F}[h]\,f)\;0\,f' \\
\mathrel{\underset{\sim}{\lhd\rhd}}\quad &\{\,\beta\text{-reduction}\,\} \\
&\lambda f \to \mathbf{let}\,\{f' = \lambda x \to \mathbf{let}\,\{x' = x + 0\}\,\mathbf{in}\,f\,x'\} \\
&\qquad\quad \mathbf{in}\,\mathbb{F}[h]\,f' \\
\mathrel{\underset{\approx}{\lhd\rhd}}\quad &\{\,x + 0 \mathrel{\underset{\approx}{\lhd\rhd}} x\,\} \\
&\lambda f \to \mathbf{let}\,\{f' = \lambda x \to \mathbf{let}\,\{x' = x\}\,\mathbf{in}\,f\,x'\} \\
&\qquad\quad \mathbf{in}\,\mathbb{F}[h]\,f' \\
\mathrel{\underset{\approx}{\lhd\rhd}}\quad &\{\,\text{variable substitution, garbage collection}\,\} \\
&\lambda f \to \mathbf{let}\,\{f' = \lambda x \to f\,x\}\,\mathbf{in}\,\mathbb{F}[h]\,f' \\
\equiv\quad &\{\,\text{definition of } \mathbb{F}\,\} \\
&\lambda f \to \mathbf{let}\,\{f' = \lambda x \to f\,x\} \\
&\qquad\quad \mathbf{in}\,(\lambda f \to \mathbf{let}\,\{f'' = \lambda x \to \mathbf{let}\,\{x' = x + 1\}\,\mathbf{in}\,f\,x\} \\
&\qquad\qquad\qquad\quad \mathbf{in}\,f\;0\;:\;h\,f'')\,f' \\
\mathrel{\underset{\sim}{\lhd\rhd}}\quad &\{\,\beta\text{-reduction}\,\} \\
&\lambda f \to \mathbf{let}\,\{f' = \lambda x \to f\,x\} \\
&\qquad\quad \mathbf{in}\,\mathbf{let}\,\{f'' = \lambda x \to \mathbf{let}\,\{x' = x + 1\}\,\mathbf{in}\,f'\,x'\} \\
&\qquad\qquad \mathbf{in}\,f'\;0\;:\;h\,f'' \\
\mathrel{\underset{\approx}{\lhd\rhd}}\quad &\{\,\text{value-}\beta\text{ on } f'\,\} \\
&\lambda f \to \mathbf{let}\,\{f'' = \lambda x \to \mathbf{let}\,\{x' = x + 1\}\,\mathbf{in}\,(\lambda x \to f\,x)\,x'\}
\end{aligned}$$

$$
\begin{array}{ll}
 & \quad\quad \mathbf{in}\ (\lambda x \to f\ x)\ 0 : h\, f'' \\
\lhd\!\rhd & \quad \{\, \beta\text{-reduction} \,\} \\
 & \lambda f \to \mathbf{let}\ \{f'' = \lambda x \to \mathbf{let}\ \{x' = x + 1\}\ \mathbf{in}\ f\ x'\} \\
 & \quad\quad \mathbf{in}\ f\ 0 : h\, f'' \\
\equiv & \quad \{\, \text{definition of } \mathbb{F} \,\} \\
 & \mathbb{F}[h]
\end{array}
$$

Now we use condition (2) to derive the new definition of *tabulate*. This requires the use of a number of the properties that we presented earlier:

$$
\begin{array}{ll}
 & \mathsf{Rep}[\checkmark \mathbb{F}[h]] \\
\equiv & \quad \{\, \text{definitions} \,\} \\
 & \lambda n \to \lambda f \to \mathbf{let}\ \{f' = \lambda x \to \mathbf{let}\ \{x' = x + n\}\ \mathbf{in}\ f\ x'\} \\
 & \quad\quad \mathbf{in}\ (\checkmark \lambda f \to \mathbf{let}\ \{f'' = \lambda x \to \mathbf{let}\ \{x'' = x + 1\}\ \mathbf{in}\ f\ x''\} \\
 & \quad\quad\quad\quad \mathbf{in}\ f\ 0 : h\, f'')\ f' \\
\lhd\!\rhd & \quad \{\, \text{tick floating } [-]\, f' \,\} \\
 & \lambda n \to \lambda f \to \mathbf{let}\ \{f' = \lambda x \to \mathbf{let}\ \{x' = x + n\}\ \mathbf{in}\ f\ x'\} \\
 & \quad\quad \mathbf{in}\checkmark (\lambda f \to \mathbf{let}\ \{f'' = \lambda x \to \mathbf{let}\ \{x'' = x + 1\}\ \mathbf{in}\ f\ x''\} \\
 & \quad\quad\quad\quad \mathbf{in}\ f\ 0 : h\, f'')\ f' \\
\lhd\!\rhd & \quad \{\, \beta\text{-reduction} \,\} \\
 & \lambda n \to \lambda f \to \mathbf{let}\ \{f' = \lambda x \to \mathbf{let}\ \{x' = x + n\}\ \mathbf{in}\ f\ x'\} \\
 & \quad\quad \mathbf{in}\checkmark \mathbf{let}\ \{f'' = \lambda x \to \mathbf{let}\ \{x'' = x + 1\}\ \mathbf{in}\ f'\ x''\} \\
 & \quad\quad\quad\quad \mathbf{in}\ f'\ 0 : h\, f'' \\
\lhd\!\rhd & \quad \{\, \text{value-}\beta \text{ on } f', \text{ garbage collection} \,\} \\
 & \lambda n \to \lambda f \to \checkmark \mathbf{let}\ \{f'' = \lambda x \to \\
 & \quad\quad\quad\quad \mathbf{let}\ \{x'' = x + 1\}\ \mathbf{in} \\
 & \quad\quad\quad\quad\quad (\checkmark \lambda x \to \mathbf{let}\ \{x' = x + n\}\ \mathbf{in}\ f\ x')\ x''\} \\
 & \quad\quad \mathbf{in}\ (\checkmark \lambda x \to \mathbf{let}\ \{x' = x + n\}\ \mathbf{in}\ f\ x')\ 0 : h\, f'' \\
\gtrsim & \quad \{\, \text{removing ticks, } \beta\text{-reduction} \,\}
\end{array}
$$

$$\begin{aligned}
&\lambda n \to \lambda f \to \checkmark\,\mathbf{let}\ \{f'' = \lambda x \to \mathbf{let}\ \{x'' = x + 1\}\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad \mathbf{let}\ \{x' = x'' + n\}\ \mathbf{in}\, f\ x'\} \\
&\qquad\qquad\qquad \mathbf{in}\ (\mathbf{let}\ \{x' = 0 + n\}\ \mathbf{in}\, f\ x')\ :\ h\, f''
\end{aligned}$$

$\lhd\!\rhd$ { associativity and identity of $+$ }

$$\begin{aligned}
&\lambda n \to \lambda f \to \checkmark\,\mathbf{let}\ \{f'' = \lambda x \to \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad \mathbf{let}\ \{x'' = x + n'\}\ \mathbf{in}\, f\ x'\} \\
&\qquad\qquad\qquad \mathbf{in}\ (\mathbf{let}\ \{x' = n\}\ \mathbf{in}\, f\ x')\ :\ h\, f''
\end{aligned}$$

$\rhd$ { variable substitution, garbage collection }

$$\begin{aligned}
&\lambda n \to \lambda f \to \checkmark\,\mathbf{let}\ \{f'' = \lambda x \to \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad \mathbf{let}\ \{x'' = x + n'\}\ \mathbf{in}\, f\ x'\} \\
&\qquad\qquad\qquad \mathbf{in}\, f\ n\ :\ h\, f''
\end{aligned}$$

$\lhd\!\rhd$ { value let-floating }

$$\begin{aligned}
&\lambda n \to \lambda f \to f\ n\ :\ \checkmark\,\mathbf{let}\ \{f'' = \lambda x \to \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in} \\
&\qquad\qquad\qquad\qquad \mathbf{let}\ \{x'' = x + n'\}\ \mathbf{in}\, f\ x'\} \\
&\qquad\qquad\qquad \mathbf{in}\, h\, f''
\end{aligned}$$

$\lhd\!\rhd$ { sums float }

$$\begin{aligned}
&\lambda n \to \lambda f \to f\ n\ :\ \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in} \\
&\qquad\qquad\qquad \checkmark\,\mathbf{let}\ \{f'' = \lambda x \to \mathbf{let}\ \{x'' = x + n'\}\ \mathbf{in}\, f\ x'\} \\
&\qquad\qquad\qquad \mathbf{in}\, h\, f''
\end{aligned}$$

$\lhd\!\rhd$ { β-expansion, tick floating }

$$\begin{aligned}
&\lambda n \to \lambda f \to f\ n\ :\ \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in} \\
&\qquad\qquad\qquad \checkmark\,(\lambda n \to \lambda f \to \\
&\qquad\qquad\qquad\qquad \mathbf{let}\ \{f'' = \lambda x \to \mathbf{let}\ \{x'' = x + n\}\ \mathbf{in}\, f\ x'\} \\
&\qquad\qquad\qquad\qquad \mathbf{in}\, h\, f'')\ n'\, f
\end{aligned}$$

$\equiv$ { definition of Rep }

$$\lambda n \to \lambda f \to f\ n\ :\ \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in}\ (\checkmark \mathsf{Rep}[h])\ n'\, f$$

$\equiv$ { taking this as our definition of $\mathbb{G}$ }

$$\mathbb{G}[\checkmark \mathsf{Rep}[h]]$$

Thus we have derived a definition of $\mathbb{G}$, from which we create the following factorised version of the program:

$$\begin{aligned} \mathit{tabulate} &= \mathbf{let}\ \{h = \mathbb{G}[h]\}\ \mathbf{in}\ \mathsf{Abs}[h] \\ \mathbb{G}[M] \quad &= \lambda n \to \lambda f \to f\ n\ :\ \mathbf{let}\ \{n' = n + 1\}\ \mathbf{in}\ M\ n'\ f \end{aligned}$$

This is the same optimised *tabulate* function that was proved correct in chapter 5. The proofs here have a similar structure to the correctness proofs from that chapter, except that we have now formalised that the new version of the *tabulate* function is indeed a time improvement of the original version. We note that the proof of (B) is complicated by the fact that η-reduction is not valid in this setting. In fact, if we assumed η-reduction then our proof of (B) here could be adapted into a proof of the assumption (A).

9.5 Conclusion

In this chapter, we have shown how improvement theory can be used to justify the worker/wrapper transformation as a program optimisation, by formally proving that, under certain natural conditions, the transformation is guaranteed to preserve or improve asymptotic time performance. This guarantee is proven with respect to an established operational semantics for call-by-need evaluation. We then verified that two examples from earlier in this thesis met the preconditions for this performance guarantee, demonstrating the use of our theory while also verifying the validity of the examples. This work is the first time that a general purpose optimisation method for lazy languages has had its effect on efficiency formally verified.

This work has a number of limitations, however. Firstly, improvement theory is untyped. As we mentioned earlier in this chapter, a typed theory would be more useful, allowing more power when reasoning about programs. This would also match more closely with the original worker/wrapper theories, which were typed. Secondly,

the theory we present here is limited to general recursion based on recursive **let** constructs. A general recursion theory can be adapted to deal with other recursion operators, but doing so may obscure the useful insights we could gain from other theories. In fact, it was adapting the original worker/wrapper theory for fix to other operators that gave rise to some of the key insights we use here. We address these two limitations in the next chapter, where we present a theory of the worker/wrapper transformation based on *bilax dinatural transformations*.

Chapter 10

Denotational Improvement

"I know the kings of England, and I quote the fights historical / from Marathon to Waterloo, in order categorical!"

— Major-General Stanley, The Pirates of Penzance

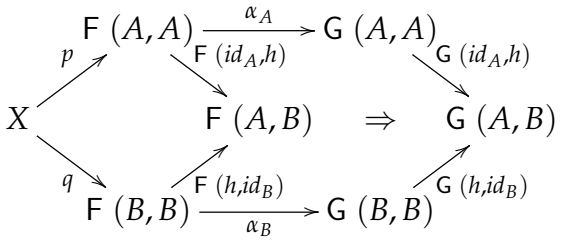
10.1 Introduction

In Chapter 6, we used category theory to generalise the correctness theory of the worker/wrapper transformation to a wide range of recursion operators. This raises the question of whether we can make a similar generalisation for the improvement side of the theory, generalising the work of the previous chapter to apply to a wider range of recursion schemes. Such a generalisation would improve on two other limitations of the original theory: it would be applicable to a wider range of resources, and would also be a *typed* theory.

If we limit ourselves to standard category theory, the answer to this question is no. After all, ordinary categories only allow arrows to be compared for equality, providing none of the structure needed to compare arrows to decide if one is "better" than another. However, *enriched* category theory can be used to give extra structure to sets of arrows. Specifically, if we enrich our categories with *preorders*, we obtain a theory that can naturally be used to compare programs.

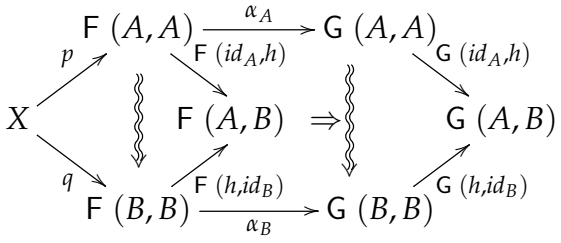
10.2 Generalising Strong Dinaturality

Recall that a family of arrows $\alpha_A : \mathsf{F}\ (A, A) \to \mathsf{G}\ (A, A)$ is *strongly dinatural* if the following diagram commutes for any $h : A \to B$:



In Chapter 6 we used this property as a generalisation of *fusion* rules, allowing us to create a generalised worker/wrapper theory applicable to any strongly dinatural recursion operator. In order to apply the same technique, we need to convert this property from an equational property to an *in*equational property that can be applied in the setting of preorder-enriched categories.

To generalise properties of categories to preorder-enriched categories, we can use the technique of *laxification*. Put simply, laxification is the process of replacing equalities with inequalities (or in the case of 2-categories, with 2-cells). By applying laxification to the earlier diagram for strong dinaturality, and drawing the inequalities $\preceq$ as a new style of arrow $\rightsquigarrow$, we obtain the following diagram for the property of *lax* strong dinaturality:



Note that F and G are now functors between preorder-enriched categories that respect the arrow ordering $\preceq$. The diagram expresses the following implication:

$$\mathsf{F}\ (id_A, h) \circ p \preceq \mathsf{F}\ (h, id_B) \circ q$$

$\Rightarrow$

$$\mathsf{G}\ (id_A, h) \circ \alpha_A \circ p \preceq \mathsf{G}\ (h, id_B) \circ \alpha_B \circ q$$

We also use the term *oplax* strong dinaturality when the ordering is reversed, i.e. when the following implication holds:

$$\mathsf{F}\ (id_A, h) \circ p \succeq \mathsf{F}\ (h, id_B) \circ q$$

$\Rightarrow$

$$\mathsf{G}\ (id_A, h) \circ \alpha_A \circ p \succeq \mathsf{G}\ (h, id_B) \circ \alpha_B \circ q$$

The choice of which direction is lax and which is oplax is arbitrary. Note that these properties do not necessarily hold in the opposite category, but lax strong dinaturality implies oplax strong dinaturality in the opposite category and vice-versa. When both lax and oplax properties hold, we describe this as *bilax* strong dinaturality. For the purposes of this chapter, we choose bilax strong dinaturality as our generalisation of strong dinaturality.

We specifically choose to use bilax strong dinaturality for three reasons. First of all, in the previous chapter we used a fusion theorem for fixed-points that bears a great deal of similarity to bilax strong dinaturality, and its bidirectionality was useful in proving the central theorem of that chapter. Secondly, Johann and Voigtländer's technique to generate inequational free theorems from polymorphic types (Johann and Voigtländer, 2004) results in precisely this same bidirectionality. Finally, unlike the other two properties, bilax strong dinaturality is self-dual.

We conclude this section by noting that bilax strong dinaturality implies an up-to-equivalence form of ordinary dinaturality. Letting $\cong$ be the conjunction of $\preceq$ and $\succeq$, i.e. $x \cong y$ iff $x \preceq y$ and $x \succeq y$, we reason as follows:

$$\mathsf{G}\ (id_A, h) \circ \alpha_A \circ \mathsf{F}\ (h, id_A) \cong \mathsf{G}\ (h, id_B) \circ \alpha_B \circ \mathsf{F}\ (id_B, h)$$

$\Leftrightarrow$ { definition of $\cong$ }

$$
\begin{array}{ll}
& \mathsf{G}\ (id_A, h) \circ \alpha_A \circ \mathsf{F}\ (h, id_A) \preceq \mathsf{G}\ (h, id_B) \circ \alpha_B \circ \mathsf{F}\ (id_B, h) \\
\wedge & \\
& \mathsf{G}\ (id_A, h) \circ \alpha_A \circ \mathsf{F}\ (h, id_A) \succeq \mathsf{G}\ (h, id_B) \circ \alpha_B \circ \mathsf{F}\ (id\_B, h) \\
\Leftarrow & \quad \{ \text{ lax and oplax strong dinaturality } \} \\
& \mathsf{F}\ (id_A, h) \circ \mathsf{F}\ (h, id_A) \preceq \mathsf{F}\ (h, id_B) \circ \mathsf{F}\ (id_B, h) \\
\wedge & \\
& \mathsf{F}\ (id_A, h) \circ \mathsf{F}\ (h, id_A) \succeq \mathsf{F}\ (h, id_B) \circ \mathsf{F}\ (id_B, h) \\
\Leftarrow & \quad \{ \text{ bifunctors and reflexivity of preorders } \} \\
& \mathit{True}
\end{array}
$$

We call this property “weakened dinaturality”.

10.3 Worker/Wrapper and Improvement, Redux

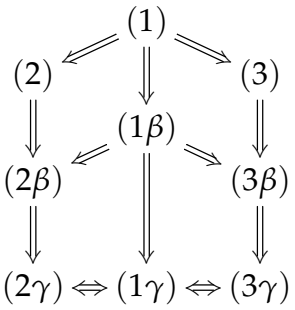
Using bilax strong dinaturality as our generalisation of strong dinaturality, we can adapt the theorem we presented in Figure 6.1 to an *inequational* version. By exchanging the ordinary category $\mathbf{Set}$ in the theorem for the preorder-enriched category $\mathbf{Ord}$ of preorders and monotonic functions, we can make ordering comparisons between the two sides of each precondition and the conclusion. The resulting theorem is presented in Figure 10.1.

Note that we relax the assumption $abs \circ rep = id$ to $abs \circ rep \cong id$ in the new theorem, as the full strength of equality is no longer required. All other equalities have been weakened to inequalities. The resulting inequalities can be interpreted as comparisons of efficiency, where $f \preceq g$ means that f is *improved* by g in terms of efficiency, i.e. ‘bigger’ in the preorder means ‘better’ in efficiency. Under this interpretation, the theorem in Figure 10.1 gives efficiency conditions under which we can factorise an original program written as $\alpha_A\ f$ into a more efficient version comprising a worker program $\alpha_B\ g$ and a wrapper function $\mathsf{G}\ (rep, abs)$.

Given:

- A preorder-enriched category $\mathcal{C}$ containing objects A and B
- Functors $\mathsf{F}, \mathsf{G} : \mathcal{C}^{op} \times \mathcal{C} \to \mathbf{Ord}$ that respect the preorder
- Arrows $abs : B \to A$ and $rep : A \to B$ in the category $\mathcal{C}$
- The assumption $abs \circ rep \cong id_A$ (where $\cong$ is the conjunction of $\preceq$ and $\succeq$)
- Elements $f \in \mathsf{F}\ (A, A)$ and $g \in \mathsf{F}\ (B, B)$
- A bilax strong dinatural transformation $\alpha : \mathsf{F} \to \mathsf{G}$

The conditions of the theorem are related as shown in the following diagram:



If any one of the following conditions holds:

(1) $g \succeq \mathsf{F}\ (abs, rep)\ f$
(1β) $\alpha_B\ g \succeq \alpha_B\ (\mathsf{F}\ (abs, rep)\ f)$
(1γ) $\mathsf{G}\ (rep, abs)\ (\alpha_B\ g) \succeq \mathsf{G}\ (rep, abs)\ (\alpha_B\ (\mathsf{F}\ (abs, rep)\ f))$

(2) $\mathsf{F}\ (rep, id)\ g \succeq \mathsf{F}\ (id, rep)\ f$
(2β) $\mathsf{G}\ (rep, id)\ (\alpha_B\ g) \succeq \mathsf{G}\ (id, rep)\ (\alpha_A\ f)$
(2γ) $\mathsf{G}\ (rep, abs)\ (\alpha_B\ g) \succeq \mathsf{G}\ (id, abs \circ rep)\ (\alpha_A\ f)$

(3) $\mathsf{F}\ (id, abs)\ g \succeq \mathsf{F}\ (abs, id)\ f$
(3β) $\mathsf{G}\ (id, abs)\ (\alpha_B\ g) \succeq \mathsf{G}\ (abs, id)\ (\alpha_A\ f)$
(3γ) $\mathsf{G}\ (rep, abs)\ (\alpha_B\ g) \succeq \mathsf{G}\ (abs \circ rep, id)\ (\alpha_A\ f)$

then we have the factorisation:

$$\alpha_A\ f \preceq \mathsf{G}\ (rep, abs)\ (\alpha_B\ g)$$

Figure 10.1: The Worker/Wrapper Theorem for Bilax Strong Dinatural Transformations

The proof of this theorem follows precisely the same form as the proof of the theorem from Chapter 6. We begin by listing the relationships between the various conditions, which follow the same pattern as before. Firstly, the (2) and (3) groups of conditions are dual, as can be seen by exchanging $\mathcal{C}$ for $\mathcal{C}^{op}$. However, the inequality is not reversed.

Secondly, each numeric condition (n) implies its corresponding $(n\beta)$ condition, which in turn implies $(n\gamma)$. The proofs are as follows:

- (1) is weakened to (1β) by applying α_B to each side.
- (2) implies (2β) by lax strong dinaturality, and (3) implies (3β) by oplax strong dinaturality.
- (1β), (2β) and (3β) can be weakened to their corresponding γ conditions by applying G (rep, abs), G (id, abs) and G (rep, id) to each side respectively.

Thirdly, condition (1) implies conditions (2) and (3), while condition (1β) implies conditions (2β) and (3β). In the first case this can be shown by simply applying F (rep, id) or F (id, abs) to both sides of condition (1). In the second case, one applies either G (rep, id) or G (id, abs) to both sides of (1β), and the result then follows from applying the weakened form of dinaturality given above.

Finally, using $abs \circ rep \cong id$, all three γ conditions are equivalent. We show this by proving that the right-hand sides are all equivalent under $\cong$. The proof for (1γ) and (2γ) is as follows:

$$
\begin{aligned}
& \mathsf{G}\ (rep, abs)\ (\alpha_B\ (\mathsf{F}\ (abs, rep)\ f)) \\
= \quad & \{\text{functors}\} \\
& \mathsf{G}\ (id, abs)\ (\mathsf{G}\ (rep, id)\ (\alpha_B\ (\mathsf{F}\ (id, rep)\ (\mathsf{F}\ (abs, id)\ f)))) \\
\cong \quad & \{\text{weakened dinaturality}\} \\
& \mathsf{G}\ (id, abs)\ (\mathsf{G}\ (id, rep)\ (\alpha_A\ (\mathsf{F}\ (rep, id)\ (\mathsf{F}\ (abs, id)\ f)))) \\
= \quad & \{\text{functors}\}
\end{aligned}
$$

$$
\begin{array}{ll}
 & \mathsf{G}\ (id, abs \circ rep)\ (\alpha_A\ (\mathsf{F}\ (abs \circ rep, id)\ f)) \\
\cong & \quad \{\, abs \circ rep \cong id \text{ implies } \mathsf{F}\ (abs \circ rep, id) \cong id \text{ because functors respect the preorder} \,\} \\
 & \mathsf{G}\ (id, abs \circ rep)\ (\alpha_A\ f)
\end{array}
$$

The proof for (1γ) and (3γ) is dual. Thus we see that all three are equivalent.

Given these relationships, it suffices to prove the theorem for one of the γ conditions. For example, it can be proved for (2γ) simply by applying the assumption $abs \circ rep = id$:

$$
\begin{array}{ll}
 & \mathsf{G}\ (rep, abs)\ (\alpha_B\ g) \\
\succeq & \quad \{\, (2\gamma) \,\} \\
 & \mathsf{G}\ (id, abs \circ rep)\ (\alpha_A\ f) \\
\cong & \quad \{\, abs \circ rep \cong id \,\} \\
 & \mathsf{G}\ (id, id)\ (\alpha_A\ f) \\
= & \quad \{\, \text{functors} \,\} \\
 & \alpha_A\ f
\end{array}
$$

This simple generalisation of our earlier theorem allows us to reason about *any* notion of improvement so long as our recursion operator treats it parametrically. We conjecture that this will be the case for a wide class of resources and operators. A proof of this will likely require a generic form of operational semantics that captures the necessary properties, perhaps akin to the globally-deterministic structural operational semantics used by Sands (1997).

By using the technique we outlined above to convert statements about preorder-enriched categories to statements about ordinary categories, we see that this new theorem for improvement is a generalisation of our earlier theorem for correctness. Thus, we have a single unified theory that covers *both* efficiency and correctness aspects of the worker/wrapper transformation.

10.4 Examples

10.4.1 Example: Least Fixed Points

In the previous chapter, we developed a theorem for the worker/wrapper transformation in the context of improvement theory à la Sands, based on general recursion. It makes sense to instantiate the theorem from this chapter in the case of general recursion, to see how the two theories compare.

To obtain such a theory, we take the same F and G as we did for least fixed points in Chapter 6:

$$\mathsf{F}\ (X, Y) = \mathbf{Cpo}\ (X, Y) \qquad\qquad \mathsf{G}\ (X, Y) = \mathsf{U}\ Y$$

In order to apply the new version of the theory to this example, we simply change the target category of these functors from $\mathbf{Set}$ to $\mathbf{Ord}$, equipping the sets with the same ordering they had in the $\mathbf{Cpo}$ setting. Thus, in the case of F, for any two elements $f, g \in \mathsf{F}\ (X, Y), f \preceq g$ if and only if $f \preceq g$ in the CPPO $X \to Y$. Likewise in the case of G, for any two elements $a, b \in \mathsf{G}\ (X, Y), a \preceq b$ if and only if $a \preceq b$ in the CPPO Y. This allows us to use our theorem to reason about the definedness of programs.

By instantiating the theorem to this case, we obtain the following preconditions:

(1) $g \succeq rep \circ f \circ abs$

(2) $g \circ rep \succeq rep \circ f$

(3) $abs \circ g \succeq f \circ abs$

(1β) $fix\ g \succeq fix\ (rep \circ f \circ abs)$

(2β) $fix\ g \succeq rep\ (fix\ f)$

(3β) $abs\ (fix\ g) \succeq fix\ f$

(1γ) $abs\ (fix\ g) \succeq abs\ (fix\ (rep \circ f \circ abs))$

(2γ) $abs\ (fix\ g) \succeq abs\ (rep\ (fix\ f))$

(3γ) $abs\ (fix\ g) \succeq fix\ f$

In this case, we only need a strictness side condition for condition (2), because in the $\succeq$ direction the fusion theorem $h \circ f \succeq g \circ h \Rightarrow h\ (\mathit{fix}\ f) \succeq \mathit{fix}\ g$ holds with no additional strictness requirements. In turn, instantiating the conclusion of our new theorem gives $\mathit{fix}\ f \preceq \mathit{abs}\ (\mathit{fix}\ g)$.

The resulting improvement theorem for fix is similar to the version from the previous chapter, with some differences. Most obviously and superficially, the theorems are different in that the previous chapter used recursive let statements rather than explicit fixed points. However, there are two more significant differences. Firstly, our new theorem is for arbitrary resource usage in the $\mathbf{Cpo}$ setting, whereas the earlier theorem was specific to time performance. Secondly, the earlier theorem had no strictness conditions, whereas the above theorem does, replacing the explicit ticks we had before. In both cases, these differences are inherited from the fusion theorem or strong dinaturality property of the underlying theory.

10.4.2 Example: Monadic Fixed Points

In Chapter 6 we gave an example instantiation for our correctness theory for *monadic* fixed points, which occur when monads are equipped with a monadic recursion operator $\mathit{mfix} : (a \to M\ a) \to M\ a$. We can apply our improvement theory to the same example, obtaining the following set of conditions:

(1) $g \succeq M\ \mathit{rep} \circ f \circ \mathit{abs}$

(2) $g \circ \mathit{rep} \succeq M\ \mathit{rep} \circ f$

(3) $M\ \mathit{abs} \circ g \succeq f \circ \mathit{abs}$

(1β) $\mathit{mfix}\ g \succeq \mathit{mfix}\ (M\ \mathit{rep} \circ f \circ \mathit{abs})$

(2β) $\mathit{mfix}\ g \succeq M\ \mathit{rep}\ (\mathit{mfix}\ f)$

(3β) $M\ \mathit{abs}\ (\mathit{mfix}\ g) \succeq \mathit{mfix}\ f$

$$(1\gamma) \quad M\ abs\ (mfix\ g) \succeq M\ abs\ (mfix\ (M\ rep \circ f \circ abs))$$

$$(2\gamma) \quad M\ abs\ (mfix\ g) \succeq M\ (abs \circ rep)\ (mfix\ f))$$

$$(3\gamma) \quad M\ abs\ (mfix\ g) \succeq mfix\ f$$

Instantiating the conclusion gives the factorisation $mfix\ f \preceq M\ abs\ (mfix\ g)$. Once again, the resulting theorem bears a close similarity with the standard *fix* theorem, and the results depend only on the bilax strong dinaturality of *mfix*. The result is a theory that allows us to reason about when factorising a monadic fixed point will result in an improved program.

As an example, consider the following monadic FRP expression, where $Sig\ \ a$ is the type of varying values of type a.

$$delay :: a \to Sig\ a \to Sig\ a$$

$$xss :: Sig\ [Int]$$

$$cat :: Sig\ [Int]$$

$$cat = mfix\ (\lambda s \to delay\ [\,]\ (xss \gg\!= \lambda xs \to return\ (s +\!\!+ xs)))$$

When these definitions are evaluated, *cat* will be a signal of lists such that the ith value of *cat* will be the concatenation of the first $i - 1$ values of xs. However, this definition is inefficient, as each step of the progression requires the entire current list to be traversed in order to append the current value of xs. We may be able to improve this by factorising *cat* into a signal that produces a so-called "difference list" in the sense of Hughes (1986), and a function that maps along that signal to turn the difference lists back into ordinary lists. Thus, our A type is $[Int]$ and our B type is $[Int] \to [Int]$. The *abs* and *rep* functions are as follows:

$$abs\ f = f\ [\,]$$

$$rep\ xs = \lambda ys \to xs +\!\!+ ys$$

The definition *cat* is the monadic fixed point of the following function:

$$
\begin{aligned}
&f :: Int \to Sig\ Int \\
&f\ s = delay\ [\,]\ (xss \gg\!= \lambda xs \to return\ (s +\!\!+ xs))
\end{aligned}
$$

We can use condition (2) to derive a function g that will serve as the core of our factorised program. This process also helps us to work out what assumptions we need to make, so that we can verify them later on.

$$
\begin{aligned}
& M\ rep \circ f \\
= \quad & \{\,\text{definitions}\,\} \\
& \lambda s \to delay\ [\,]\ (xss \gg\!= \lambda xs \to return\ (s +\!\!+ xs)) \\
& \quad \gg\!= (\lambda rs \to return\ (\lambda ys \to rs +\!\!+ ys)) \\
\cong \quad & \{\,\text{assumption 1: } delay\ x\ xm \gg\!= f \cong f\ x \gg\!= \lambda y \to delay\ y\ (xm \gg\!= f)\,\} \\
& \lambda s \to return\ (\lambda ys \to [\,] +\!\!+ ys) \\
& \quad \gg\!= \lambda y \to delay\ y\ (xm \gg\!= \lambda rs \to return\ (\lambda ys \to rs +\!\!+ ys)) \\
= \quad & \{\,\text{monad laws, } ([\,]+\!\!+) \text{ is identity}\,\} \\
& \lambda s \to delay\ id\ (xss \gg\!= \lambda xs \to return\ ((s +\!\!+ xs)+\!\!+)) \\
\preceq \quad & \{\,\text{assumption 2: } ((x +\!\!+ y)+\!\!+) \preceq (x+\!\!+) \circ (y+\!\!+)\,\} \\
& \lambda s \to delay\ id\ (xss \gg\!= \lambda xs \to return\ (rep\ s \circ (xs+\!\!+))) \\
= \quad & \{\,\text{definition of } rep\,\} \\
& \lambda s \to delay\ id\ (xss \gg\!= \lambda xs \to return\ ((s+\!\!+) \circ (xs+\!\!+))) \\
= \quad & \{\,\text{let } g\ s = delay\ id\ (xss \gg\!= \lambda xs \to return\ (s \circ (xs+\!\!+)))\,\} \\
& g \circ rep
\end{aligned}
$$

The resulting definition of g can be used to produce the following program:

$$
\begin{aligned}
&work :: Sig\ ([Int] \to [Int]) \\
&work = mfix\ (\lambda s \to delay\ id\ (xss \gg\!= \lambda xs \to return\ (s \circ (xs+\!\!+)))) \\
&cat = \mathbf{do}\ r \leftarrow work \\
&\qquad\qquad return\ (abs\ r)
\end{aligned}
$$

In this reasoning, we have made some assumptions. Firstly, we have assumed that $delay\ x\ xm \gg\!= f \cong f\ x \gg\!= \lambda y \to delay\ y\ (xm \gg\!= f)$, which is a requirement

that $delay$ is in some sense well-behaved with respect to efficiency. Secondly, we have assumed that $((x +\!\!+ y) +\!\!+) \preceq (x +\!\!+) \circ (y +\!\!+)$, which we can verify as follows:

$$
\begin{aligned}
& ((x +\!\!+ y) +\!\!+) \\
= \quad & \{ \text{ operator sections } \} \\
& \lambda z \to (x +\!\!+ y) +\!\!+ z \\
\preceq \quad & \{ \text{ right-association is more efficient than left-association } \} \\
& \lambda z \to x +\!\!+ (y +\!\!+ z) \\
\cong \quad & \{ \text{ reverse beta-reduction } \} \\
& \lambda z \to x +\!\!+ (\lambda w \to y +\!\!+ w)\ z \\
= \quad & \{ \text{ operator sections, function composition } \} \\
& ((x +\!\!+) \circ (y +\!\!+))
\end{aligned}
$$

Therefore, so long as right-association is more efficient than left-association (which we know to be true for the standard Haskell append) this assumption holds. We can thus conclude that, so long as our $mfix$ operator is bilax strong dinatural and so long as $delay$ behaves as we expect with regard to efficiency, the factorised program is faster.

10.5 Remarks

The process of generalising from the correctness theorem of Figure 6.1 to the improvement theorem of Figure 10.1 was entirely straightforward, our treatment of correctness leading immediately to a related treatment of improvement. This is encouraging, as it helps to justify our choice of machinery. Furthermore, the similarities between the two theorems mean that it should be straightforward to adapt a proof of correctness into a proof of improvement, a benefit this work shares with the work of the previous chapter. However, in this work the correctness theorem can be considered a specialisation of the improvement theorem, which serves as progress toward bridging the gap between correctness and efficiency.

Chapter 11

Summary and Evaluation

> *"If every pork chop were perfect, we wouldn't have hot dogs."*
>
> *— Greg Universe, Steven Universe*

In Part III, we presented two theories of program improvement for the worker/wrapper transformation. These theories had some commonalities with each other, as well as with the correctness theories presented in the previous part. Neither theory is strictly stronger than the other, although a theory similar to the first can be derived as an instantiation of the second.

The first improvement theory was based on general recursion, modelling recursive programs in an operational semantics with recursive let statements. As was the case for the correctness theory for fixed points, in one sense this has a high degree of generality as other recursion schemes can be implemented using general recursion. However, the same problems of overhead and restricted setting also apply, with this theory restricting us to a specific operational semantics. This theory also lacks a type system, meaning that the user of the theory cannot use properties derived from types when applying it. Finally, while this theory has separate (A), (B) and (C) assumptions corresponding to the selection of assumptions of the correctness theories for fix and unfold, assumption (A) is unsatisfiable in this context and only included for complete-

ness. A typed theory may allow assumption (A) to be satisfiable again.

The second improvement theory was based on bilax strong dinatural transformations. This is a generalisation of the previous correctness theory for strong dinatural transformations and inherits the same degree of generality, being applicable to a wide variety of recursion operators so long as they satisfy the underlying assumption of bilax strong dinaturality. It also inherits the same limitations, lacking separate (A), (B) and (C) conditions and requiring an ad hoc approach to strictness. Furthermore, proving bilax strong dinaturality of an operator will be more difficult than merely proving strong dinaturality in most cases, the main exception being when the ordering is discrete (and the properties therefore coincide).

Part IV: Conclusion

Wrapping it up

Chapter 12

Summary and Further Work

"It's the end... but the moment has been prepared for."

— The Fourth Doctor, Logopolis

In this thesis, we studied the dual aspects of correctness and improvement for a general program optimisation, the worker/wrapper transformation. We developed new general theories of this transformation, and showed that these theories were all linked by the same structure centered around an application of one of the related properties of rolling, fusion or (bilax) strong dinaturality. This showed that the transformation could be unified along several directions, ultimately leading to a general worker/wrapper theory that could be applied to both improvement and correctness. Specifically, we made the following contributions:

- In Chapter 5 we gave a novel presentation of the worker/wrapper transformation for programs written in the form of an unfold. This presentation bore a great deal of similarity to existing presentations given for folds and fixed points, but was different in that it could be applied in the setting of CPPOs without the need for strictness conditions. Furthermore, we were able to develop a modified version of the (C) assumption that was not applicable in the fix case.
- In Chapter 6 we demonstrated that the core of existing worker/wrapper theo-

ries was the application of either the rolling rule or fusion, and used this to develop a general worker/wrapper transformation based on the categorical notion of strong dinaturality. This form of the transformation could be applied to any operator satisfying the property of strong dinaturality, which in many cases follows from parametricity, and the correctness proof for this transformation followed the same structure as existing correctness proofs. We showed how several other worker/wrapper theories could be obtained by instantiating this one, including the previous theories for fixed points and unfolds.

- In Chapter 9 we used improvement theory to verify that, under certain natural conditions, the worker/wrapper transformation for least fixed points does not make programs slower. This was the first case of such a result being proved for a general-purpose program optimisation, and demonstrates the utility of both the worker/wrapper transformation and improvement theory.

- In Chapter 8 we introduced the idea of using preorder-enriched categories to reason about notions of program improvement, which we subsequently used in Chapter 10 to extend the presentation of the worker/wrapper transformation based on strong dinaturality to one based on *bilax* strong dinaturality. The result was a generalised theory of improvement for the worker/wrapper transformation, giving improvement properties for any recursion operator satisfying its preconditions. This theory was strictly stronger than the theory for strong dinaturality, as this theory could be recovered simply by choosing the ordering in which program improvement and program equivalence coincide.

12.1 Conclusion

This thesis presents five different theorems, each one giving correctness or improvement properties for the worker/wrapper transformation. These theorems have many

commonalities: all five are factorisation theorems for recursive programs, all have proofs with the same fundamental structure centred on applications of either fusion or rolling rules and all provide a range of conditions that can be used to derive the improved program. However these theorems differ by the settings and operators they are applicable to as well as whether they prove correctness or improvement. We summarise the five theorems in the following table:

Type	Correctness			Improvement	
Theorem	Fig. 4.1	Fig. 5.2	Fig. 6.1	Fig. 9.2	Fig. 10.1
Setting	CPPOs	Category Theory	Category Theory	Sestoft Abstract Machine	Enriched Category Theory
Operators	*fix*	*unfold*	strong dinaturals	recursive bindings	bilax strong dinaturals

For the rest of this section we shall discuss some important differences between these five theorems.

We have two theorems covering recursive bindings, one based on least fixed points and one based on recursive bindings. However, while generally recursive programs can be modelled equally well by both of these, the two theorems cannot be unified because of their different settings. The correctness theorem for *fix* has a denotational setting of CPPOs, while the improvement theorem for recursive bindings is based on an operational semantics. While CPPOs are a natural setting for reasoning about the correctness of lazy functional programs, an operational setting was needed to reason about time costs.

The correctness theorems for unfolds and strong dinaturals are both in the same setting of category theory. However, once again these theorems are not perfectly unifiable. The unfold theorem has a choice of assumptions (A), (B) and (C) for the relationship between *abs* and *rep*, while the strong dinaturality theorem offers only one such option. While it is possible to generate an unfold theorem by instantiating the strong dinaturality theorem, it is quite as strong as the original unfold theorem.

Finally, the first improvement theory is based on an operational semantics, and the second is based in category theory and thus geared more toward denotational semantics. Both are general, as the operational improvement theory is based on general recursion which can be used to implement any recursion scheme while the categorical theory allows for a wide range of recursion operators. However, in order to use the categorical theorem it is necessary to prove that the recursion operator and the relevant notion of improvement satisfy the assumptions of the theory. The operational theory is more immediately applicable, but can only be used to reason about time costs in a call-by-need setting.

12.2 Relation to Other Work

Because of the generality of the dinaturality-based theories, it is informative to consider how they relate to other worker/wrapper-style techniques.

The bilax dinaturality-based worker/wrapper theory can be applied to Milner's theory of program simulation (Milner, 1971). The operation that takes a state machine and produces the associated partial function is bilax strong dinatural in the category of relations from the functor $\lambda X\ Y\ Z\ .\ X + Y \leftrightarrow Y + Z$ to $\lambda X\ Y\ Z\ .\ X \leftrightarrow Z$. Given state machines $S\ :\ A + B \leftrightarrow B + C$, $T\ :\ A' + B' \leftrightarrow B' + C'$ and relations $R1\ :\ A \leftrightarrow A'$, $R2\ :\ B \leftrightarrow B'$, $R3\ :\ C \leftrightarrow C'$, lax strong dinaturality gives us that if $(R2 + R3) \circ T \preceq S \circ (R1 + R2)$ then $R3 \circ \mathit{func}\ (T) \preceq \mathit{func}\ (S) \circ R1$, where $\preceq$ is the inclusion of relations. This implication is precisely Milner's theorem 3.3(i). The oplax case follows from duality, as all arrows in the category of relations are reversible.

The dinaturality theories can be seen as a functional analogue to Hoare's work on abstract data (Hoare, 1972). We can consider the abstract program as a dinatural transformation: it takes a concrete type A and a set of operations on that type, and produces a program that uses that type in place of the original abstract type. The worker/wrapper theory then tells us when it is safe to replace one implementation

with another, providing a wide variety of conditions we can use to verify this safety.

We cannot yet apply the worker/wrapper theory to the case of defunctionalisation (Reynolds, 1998a), as while it is straightforward to use the worker/wrapper theory to move to a larger type, there is not yet a good story for moving to a *smaller* type. The condition (C) allows us to do this in some theories by weakening the requirement that $abs \circ rep = id$ (and it is this requirement that implies that the new type B is larger than the original type A), but as yet there is no equivalent condition for either of the two dinaturality-based theories.

12.3 Further Work

There are a number of potential avenues for further work, which we outline here. For convenience, we split these ideas into the two categories of theory and practice.

12.3.1 Theory

Our generalised theory based on (bilax) strong dinaturality lacks the separate (A), (B) and (C) assumptions of previous theories. This flaw is minor, as the assumption (A) is by far the most used assumption in practice, but it is still worth seeing if this could be rectified. Not only would this make the strong dinaturality theory a true generalisation of previous theories, it would also give us further insight into the worker/wrapper transformation in general.

Complementing the previous point, further worker/wrapper theories could be developed for other recursion operators or settings. Comparing these theories to the corresponding instantiations of the generalised theory would point to ways we could improve the generalised theory. Furthermore, specific theories will be generally easier to apply than generalised ones, as they require less knowledge of the underlying theory. By the same token, it may be worth developing more specific worker/wrapper improvement theories for recursion patterns other than general recursion.

In developing our generalised theories, we made the assumption that all operators of interest would be (bilax) strong dinatural, and justified this with an appeal to parametricity. However, there are some cases where parametricity is not strong enough to fulfil the requirement that an operator be strongly dinatural. Thus, we should investigate the relationship between strong dinaturality and parametricity more closely and use our findings to refine our generalise worker/wrapper theory, either by adding more requirements or by modifying our assumption of strong dinaturality to better fit what is provided by parametricity.

The only worker/wrapper theory for improvement tied to a concrete operational semantics is limited to reasoning about time efficiency. Gustavsson and Sands (1999, 2001) have developed an improvement theory for space, so this would be an obvious next step. More generally, we could apply a technique such as that used by Sands (1997) to develop a theory that applies to a large class of resources, and examine which assumptions must be made about the resources we consider for our theory to apply. Such a theory would put our categorical improvement theory on a more solid footing.

In Chapter 9, we briefly discussed the potential of a typed improvement theory. While this is partially fulfilled by our worker/wrapper theory based on bilax strong dinaturality, we may also benefit from developing an *operational* typed theory. This could be used to justify the assumptions made by the dinaturality theory as well as being a useful theory in its own right.

At the moment, in order to verify preconditions for our improvement theorems, one would need to fall back on pre-existing theories of improvement based on the operational semantics of a programming language. This is undesirable, as it increases the amount of theoretical knowledge and proof skills that a programmer would need to use our theory. We would like to develop a purely categorical theory of improvement, allowing improvement relations to be proved in a purely abstract way without the need to reason at the level of an underlying concrete semantics.

Finally, in all cases, our assumed relationship between abs and rep is some kind of equivalence. In the context of improvements, this implies that $abs\ (rep\ x)$ cannot be more than a constant factor slower than x, which limits our choice for these operations. We would like to also be able to cover cases where abs and rep are expensive, but the extra cost is made up for by the overall efficiency gain of the transformation. To cover such cases, we would require a richer version of improvement theory that is able to quantify *how much* one program is better than another.

12.3.2 Practice

As our examples show, the calculations required to derive an improved program can often be quite involved. This is especially the case when applying improvement theory, where much of the work involves the careful bookkeeping of ticks. The HERMIT system, devised by a team at the University of Kansas (Farmer et al., 2012; Sculthorpe et al., 2013), facilitates program transformations by providing an interactive interface for program transformation that verifies correctness. If improvement theory could be integrated into such a system, it would be significantly easier to apply our new improvement-based worker/wrapper theories.

Finally, we would like to investigate the potential for automating the transformation. By far the biggest hurdle for this would be automating the verification of the preconditions. We believe that the best approach to doing this would be to adapt algorithms designed for *higher-order unification* (Huet, 1975), the problem of solving equations on lambda terms. It may even be possible to adapt these algorithms to deal with the *inequational* conditions of our improvement theorems. It may be possible to integrate higher-order unification into either HERMIT or GHC to facilitate the automation of the worker/wrapper transformation.

Bibliography

Anon. The Descent of Inanna, C. 3500–1900 BCE.

Ralph-Johan Back and Joakim Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

Edwin S. Bainbridge, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. Functorial Polymorphism. *Theoretical Computer Science*, 70(1):35–64, 1990.

Hendrik Pieter Barendregt. *The Lambda Calculus: its syntax and semantics*. North-Holland Amsterdam, 1984.

Richard Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2nd edition, 1998.

Edwin Brady. IDRIS: Systems Programming Meets Full Dependent Types. In *Programming Languages meets Program Verification*, pages 43–54. ACM, 2011.

Thierry Coquand. Infinite Objects in Type Theory. In *TYPES '93*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 1993.

Louis de Broglie. *Recherches sur la théorie des quanta (Researches on quantum theory)*. PhD thesis, Faculty of Sciences, University of Paris, 1924.

A. de Bruin and E. P. de Vink. Retractions in Comparing Prolog Semantics. In *Mathematical Foundations of Computer Science*, pages 180–186. Springer-Verlag, 1990.

René Descartes. *Meditationes de prima philosophia, in qua Dei existentia et animæ immortalitas demonstratur (Mediations on first philosophy, in which the existence of God and the immortality of the soul are demonstrated)*. 1641.

Samuel Eilenberg and Saunders Mac Lane. General Theory of Natural Equivalences. *Transactions of the American Mathematical Society*, 58(2):231–294, 1945.

Levent Erkök and John Launchbury. Recursive Monadic Bindings. In *International Conference on Functional Programming*, pages 174–185. ACM, 2000.

Andrew Farmer, Andy Gill, Ed Komp, and Neil Sculthorpe. The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. In *Haskell Symposium*, pages 1–12. ACM, 2012.

Neil Ghani, Tarmo Uustalu, and Varmo Vene. Build, Augment and Destroy, Universally. In *Programming Languages and Systems: Second Asian Symposium*, pages 327–347. Springer, 2004.

Andy Gill and Graham Hutton. The Worker/Wrapper Transformation. *Journal of Functional Programming*, 19(2):227–251, 2009.

Jörgen Gustavsson and David Sands. A Foundation for Space-Safe Transformations of Call-by-Need Programs. *Electronic Notes on Theoretical Computer Science*, 26: 69–86, 1999.

Jörgen Gustavsson and David Sands. Possibilities and Limitations of Call-by-Need Space Improvement. In *International Conference on Functional Programming*, pages 265–276. ACM, 2001.

Jennifer Hackett and Graham Hutton. Worker/Wrapper/Makes It/Faster. In *International Conference on Functional Programming*, pages 95–107. ACM, 2014.

Jennifer Hackett and Graham Hutton. Programs for Cheap! In *Logic in Computer Science*, pages 115–126. IEEE, 2015.

Jennifer Hackett, Graham Hutton, and Mauro Jaskelioff. The Under-Performing Unfold: A New Approach to Optimising Corecursive Programs. In *Symposium on Implementation and Application of Functional Languages*, pages 4321–4332. ACM, 2013.

Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, Department of Computer Science, University of Edinburgh, 1987.

Kevin Hammond and Greg Michaelson. Hume: A Domain-Specific Language for Real-Time Embedded Systems. In *Generative Programming and Component Engineering*, pages 37–56. Springer, 2003.

Matthew Hennessy and Robin Milner. On Observing Nondeterminism and Concurrency. In *Automata, Languages and Programming, 7th Colloquium*, pages 299–309. Springer, 1980.

C. A. Hoare. Proof of Correctness of Data Representations. *Acta Informatica*, 1(4): 271–281, 1972. ISSN 0001-5903.

C. A. R. Hoare and Jifeng He. *Data Refinement in a Categorical Setting*. Oxford University Computing Laboratory, 1990.

Martin Hofmann and Steffen Jost. Static Prediction of Heap Space Usage for First-Order Functional Programs. In *Principles of Programming Languages*, pages 185–197. ACM, 2003.

Martin Hofmann and Georg Moser. Amortised Resource Analysis and Typed Polynomial Interpretations. In *Rewriting Techniques and Applications—Typed Lambda Calculi and Applications*, pages 272–286, 2014.

Catherine Hope. *A Functional Semantics for Space and Time*. PhD thesis, School of Computer Science, University of Nottingham, 2008.

Gérard P. Huet. A Unification Algorithm for Typed λ-Calculus. *Theoretical Computer Science*, 1(1):27–57, 1975.

John Hughes. A Novel Representation of Lists and its Application to the Function “reverse". *Information Processing Letters*, 22(3):141–144, 1986.

John Hughes. Generalising Monads to Arrows. *Science of Computer Programming*, 37 (1-3):67–111, 2000.

John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems using Sized Types. In *Principles of Programming Languages*, pages 410–423. ACM, 1996.

Graham Hutton. *Programming in Haskell*. Cambridge University Press, September 2016.

Graham Hutton and Jeremy Gibbons. "the generic approximation lemma". *Information Processing Letters*, 79(4):197–201, 2001.

Graham Hutton, Mauro Jaskelioff, and Andy Gill. Factorising Folds for Faster Functions. *Journal of Functional Programming Special Issue on Generic Programming*, 20(3&4):353–373, 2010.

Patricia Johann and Janis Voigtländer. Free Theorems in the Presence of *seq*. In *Principles of Programming Languages*, pages 99–110, 2004.

Gilles Kahn. Natural Semantics. In *Symposium on Theoretical Aspects of Computer Science*, pages 22–39. Springer, 1987.

Gregory M. Kelly. Basic Concepts of Enriched Category Theory. In *LMS Lecture Notes*, volume 64. Cambridge University Press, 1982.

Stephen Cole Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

Donald E. Knuth. Notes on the van Emde Boas construction of priority deques: an instructive use of recursion, 1977. Classroom notes, Stanford University.

Joachim Lambek. From λ-calculus to Cartesian Closed Categories. pages 375–402. Academic Press, 1980.

Saunders Mac Lane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer Science & Business Media, 1971.

Grant R. Malcolm. *Algebraic Data Types and Program Transformation*. PhD thesis, Rijksuniversiteit Groningen, 1990.

John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3:184–195, 1960.

Erik Meijer, Maarten M. Fokkinga, and Ross Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture*, volume 523, pages 124–144. Springer, 1991.

Robin Milner. An Algebraic Definition of Simulation Between Programs. In *International Joint Conference on Artificial Intelligence*, pages 481–489. Morgan Kaufmann, 1971.

Andrew Moran and David Sands. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. 1999a. Extended version of Moran and Sands (1999b), available at `http://tinyurl.com/ohuv8ox`.

Andrew Moran and David Sands. Improvement in a Lazy Context: An Operational Theory for Call-by-Need. In *Principles of Programming Languages*, pages 43–57. ACM, 1999b.

Philip S. Mulry. Categorical Fixed Point Semantics. *Theoretical Computer Science*, 70 (1):85–97, 1990.

Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1999.

David Michael Ritchie Park. Concurrency and Automata on Infinite Sequences. In *Theoretical Computer Science*, pages 167–183. Springer, 1981.

Will Partain. The nofib Benchmark Suite of Haskell Programs. In *Glasgow Workshop on Functional Programming*, pages 195–202. Springer, 1992.

Ross Paterson. A New Notation for Arrows. In *International Conference on Functional Programming*, pages 229–240. ACM, 2001.

Simon L. Peyton Jones. Compiling Haskell by Program Transformation: A Report from the Trenches. In *European Symposium on Programming*, pages 18–44. Springer, 1996.

Simon L. Peyton Jones and John Launchbury. Unboxed Values as First Class Citizens in a Non-Strict Functional Language. In *Functional Programming Languages and Computer Architecture*, pages 636–666. Springer, 1991.

Gordon Plotkin and Martín Abadi. A Logic for Parametric Polymorphism. In *Typed Lambda Calculi and Applications*, pages 361–375. Springer, 1993.

Gordon D. Plotkin. A Structural Approach to Operational Semantics. *Journal of Logic and Algebraic Programming*, 60:3–15, 2004.

John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998a.

John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998b.

Jan J. M. M. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, 249(1):3–80, 2000.

David Sands. Operational Theories of Improvement in Functional Languages. In *Glasgow Workshop on Functional Programming*, pages 298–311. Springer, 1991.

David Sands. From SOS Rules to Proof Principles: An Operational Metatheory for Functional Languages. In *Principles of Programming Languages*, pages 428–441. ACM, 1997.

Patrick M. Sansom and Simon L. Peyton Jones. Formally Based Profiling for Higher-Order Functional Languages. *Transactions on Programming Languages and Systems*, 19(2), 1997.

Dana Scott. *Outline of a Mathematical Theory of Computation*. Oxford University Computing Laboratory, Programming Research Group, 1970.

Dana Scott. Domains for Denotational Semantics. In *Automata, Languages and Programming*, volume 140, pages 577–610. Springer, 1982.

Dana Scott and Christopher Strachey. *Toward a Mathematical Semantics for Computer Languages*. Oxford University Computing Laboratory, Programming Research Group, 1971.

Neil Sculthorpe and Graham Hutton. Work It, Wrap It, Fix It, Fold It. *Journal of Functional Programming*, 24(1):113–127, 2014.

Neil Sculthorpe, Andrew Farmer, and Andy Gill. The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language. In *Implementation and Application of Functional Languages*, pages 86–103, 2013.

Peter Sestoft. Deriving a Lazy Abstract Machine. *Journal of Functional Programming*, 7(3):231–264, 1997.

David A. Turner. Elementary Strong Functional Programming. In *Functional Programming Languages in Education*, pages 1–13. Springer, 1995.

Tarmo Uustalu. A Note on Strong Dinaturality, Initial Algebras and Uniform Parameterized Fixpoint Operators. In *Fixed Points in Computer Science*, 2010.

Pedro B. Vasconcelos and Kevin Hammond. Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Programs. In *Implementation of Functional Languages*, pages 86–101. Springer, 2003.

Philip Wadler. Theorems for Free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM, 1989.

Philip Wadler. The Essence of Functional Programming. In *Principles of Programming Languages*, pages 1–14. ACM, 1992.

Glynn Winskel. *The Formal Semantics of Programming Languages — An Introduction*. MIT press, 1993.