Work It, Wrap It, Fix It, Fold It (Extended Version)

NEIL SCULTHORPE
University of Kansas, USA

GRAHAM HUTTON
University of Nottingham, UK

Abstract

The worker/wrapper transformation is a general-purpose technique for refactoring recursive programs to improve their performance. The two previous approaches to formalising the technique were based upon different recursion operators and different correctness conditions. In this article we show how these two approaches can be generalised in a uniform manner by combining their correctness conditions, extend the theory with new conditions that are both necessary and sufficient to ensure the correctness of the worker/wrapper technique, and explore the benefits that result. All the proofs have been mechanically verified using the Agda system.

1 Introduction

A fundamental objective in computer science is the development of programs that are clear, efficient and correct. However, these aims are often in conflict. In particular, programs that are written for clarity may not be efficient, while programs that are written for efficiency may be difficult to comprehend and contain subtle bugs. One approach to resolving these tensions is to use *program transformation* techniques to systematically rewrite programs to improve their efficiency, without compromising their correctness.

The focus of this article is the *worker/wrapper transformation*, a transformation technique for improving the performance of recursive programs by using more efficient intermediate data structures. The basic idea is simple and general: given a recursive program of some type A, we aim to factorise it into a more efficient *worker* program of some other type B, together with a *wrapper* function of type $B \to A$ that allows the new worker to be used in the same context as the original recursive program.

Special cases of the worker/wrapper transformation have been used for many years. For example, the technique has played a key role in the Glasgow Haskell Compiler since its inception more than twenty years ago, to replace the use of boxed data structures by more efficient unboxed data structures (Peyton Jones & Launchbury, 1991). However, it is only recently — in two articles that lay the foundations for the present work (Gill & Hutton, 2009; Hutton *et al.*, 2010) — that the worker/wrapper transformation has been formalised, and considered as a general approach to program optimisation.

The original formalisation (2009) was based upon a least-fixed-point semantics of recursive programs. Within this setting the worker/wrapper transformation was explained and formalised, proved correct, and a range of programming applications presented. Three key benefits of the technique were also identified:

- It provides a general and systematic approach to transforming a recursive program of one type into an equivalent program of another type.
- It is straightforward to understand and apply, requiring only basic equational reasoning techniques, and often avoiding the need for induction.
- It allows many optimisation techniques that at first sight may seem to be unrelated to be captured within a single unified framework.

Using fixed points allowed the worker/wrapper transformation to be formalised, but did not take advantage of the additional structure that is present in many recursive programs. To this end, a more structured approach (2010) was then developed based upon initial-algebra semantics, a categorical approach to recursion that is widely used in program optimisation (Bird & de Moor, 1997). More specifically, a worker/wrapper theory was developed for programs defined using fold operators, which encapsulate a common pattern of recursive programming. In practice, using fold operators results in simpler transformations than the approach based upon fixed points. Moreover, it also admitted the first formal proof of correctness of a new approach (Voigtländer, 2008) to optimising monadic programs.

While the two previous articles were nominally about the same technique, they were quite different in their categorical foundations and correctness conditions. The first was founded upon least fixed points in the category **CPO** of complete partial orders and continuous functions, and identified a hierarchy of conditions on the conversion functions between the original and worker types that are sufficient to ensure correctness. In contrast, the second was founded upon initial algebras in an arbitrary category $\mathbb{C}$, and identified a lattice of sufficient correctness conditions on the original and worker algebras. This raises the question of whether it is possible to combine or unify the two different approaches. The purpose of this new article is to show how this can be achieved, and to explore the benefits that result. More precisely, the article makes the following contributions:

- We show how the least-fixed-point and initial-algebra approaches to the worker/wrapper transformation can be generalised in a uniform manner by combining their different sets of correctness conditions (sections 3 and 5).
- We identify necessary conditions for the correctness of the worker/wrapper technique, in addition to the existing sufficient conditions, thereby ensuring that the theory is as widely applicable as possible[1] (sections 3 and 5).
- We use our new theory to develop a specialised worker/wrapper theory for folds in **CPO** that eliminates all unnecessary strictness conditions (section 6).

The practical application of our new theory is illustrated using a series of new examples, and all the examples from the previous articles are still valid in our generalised framework. The article is aimed at readers who are familiar with the basics of programming-language

[1] Specifically, we identify conditions that are necessary and sufficient to ensure that the worker/wrapper factorisation and fusion properties are both valid.

semantics, in particular the least-fixed-point (Schmidt, 1986) and initial-algebra (Bird & de Moor, 1997) approaches, but all necessary concepts and results are reviewed. No previous experience with the worker/wrapper transformation is assumed, as this extended version of the article is intended to subsume Gill & Hutton (2009) and Hutton *et al.* (2010). A condensed version of this article that omits the examples and proofs is published in the Journal of Functional Programming (JFP), and a mechanical verification of the proofs in Agda is available as supplementary material on the JFP website.

2 Least-Fixed-Point Semantics

The original formalisation of the worker/wrapper transformation was based on a least-fixed-point semantics of recursion, in a domain-theoretic setting in which programs are continuous functions on complete partial orders. In this section we review some of the basic definitions and properties from this approach to program semantics, and introduce our notation. For further details, see for example Schmidt (1986).

2.1 Basic Definitions

A *complete partial order* (cpo)[2] is a set with a partial-ordering $\sqsubseteq$, a least element $\bot$, and limits $\bigsqcup$ (least upper bounds) of all non-empty chains. A function f between cpos is *continuous* if it is monotonic, i.e. $x \sqsubseteq y \Rightarrow f\ x \sqsubseteq f\ y$ for all values x and y, and preserves the limit structure, i.e. $f(\bigsqcup X) = \bigsqcup \{f\ x \mid x \in X\}$ for all non-empty chains X. If it also preserves the least element, i.e. $f\ \bot = \bot$, the function is *strict*.

2.2 Least Fixed Points

A *fixed point* of a function f is a value x for which $f\ x = x$. Kleene's well-known fixed-point theorem (Schmidt, 1986) states that any continuous function f on a cpo has a least fixed point, denoted by $\mathit{fix}\, f$, which can be constructed as the limit of the infinite chain

$$\bot \sqsubseteq f\ \bot \sqsubseteq f\ (f\ \bot) \sqsubseteq f\ (f\ (f\ \bot)) \sqsubseteq f\ (f\ (f\ (f\ \bot))) \sqsubseteq \ldots$$

and is characterised by the following two properties (Backhouse, 2002):

Lemma 2.1 (*Computation*)

$$\mathit{fix}\, f\ =\ f\ (\mathit{fix}\, f)$$

Lemma 2.2 (*Induction*)

$$f\ x \sqsubseteq x \Rightarrow \mathit{fix}\, f \sqsubseteq x$$

The first property states that $\mathit{fix}\, f$ is a fixed point of the function f, while the second states that $\mathit{fix}\, f$ is the least *prefix* point of f, i.e. the least value x for which $f\ x \sqsubseteq x$. The second property is equivalent to being the least fixed point of f, but the above formulation turns out to be more useful for reasoning purposes.

[2] The notion of complete partial order that we use is technically that of a pointed ω-cpo, but we will use the simpler term cpo throughout this article.

2.3 Useful Properties

The basic proof technique for least fixed points is *fixed-point induction* (Winskel, 1993). Suppose that f is a continuous function on a cpo and that P is a *chain-complete* predicate on the same cpo, i.e. whenever the predicate holds for all elements in a non-empty chain then it also holds for the limit of the chain. Then fixed-point induction states that if the predicate holds for the least element of the cpo (the base case) and is preserved by the function f (the inductive case), then it also holds for $\mathit{fix}\, f$:

Lemma 2.3 (*Fixed-Point Induction*)
If P is chain-complete, then:

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

Proof
See Winskel (1993). □

Fixed-point induction can be used to verify the well-known *fixed-point fusion* property (Meijer *et al.*, 1991), which states that the application of a function to a *fix* can be re-expressed as a single *fix*, provided that the function is strict and satisfies a simple commutativity condition with respect to the *fix* arguments:

Lemma 2.4 (*Fixed-Point Fusion*)

$$f \circ g\ =\ h \circ f\ \wedge\ \mathrm{strict}\, f\ \ \Rightarrow\ \ f\ (\mathit{fix}\, g)\ =\ \mathit{fix}\, h$$

Proof

$$\begin{array}{ll} & f\ (\mathit{fix}\, g)\ =\ \mathit{fix}\, h \\ \Leftrightarrow & \qquad \{\ \text{define } P\ (a,b)\ \Leftrightarrow\ f\ a\ =\ b\ \} \\ & P\ (\mathit{fix}\, g, \mathit{fix}\, h) \\ \Leftarrow & \qquad \{\ \text{fixed point induction}\ \} \\ & P\ (\bot,\bot)\ \wedge\ (\forall\, a,b.\, P\ (a,b)\ \Rightarrow\ P\ (g\ a, h\ b)) \end{array}$$

The first conjunct simplifies to $f\ \bot\ =\ \bot$; the second can be simplified as follows:

$$\begin{array}{ll} & \forall\, a,b.\, P\ (a,b)\ \Rightarrow\ \ P\ (g\ a, h\ b) \\ \Leftrightarrow & \qquad \{\ \text{applying } P\ \} \\ & \forall\, a,b.\, f\ a\ =\ b\ \Rightarrow\ f\ (g\ a)\ =\ h\ b \\ \Leftrightarrow & \qquad \{\ \text{simplification}\ \} \\ & \forall\, a.\, f\ (g\ a)\ =\ h\ (f\ a) \\ \Leftrightarrow & \qquad \{\ \text{unapplying } \circ, \text{extensionality}\ \} \\ & f \circ g\ =\ h \circ f \end{array}$$

□

Note that if the strictness condition on the function f was dropped, then $f\ (\mathit{fix}\, g)$ would still be a fixed point of h, but not necessarily the least fixed point.

Finally, a key property of *fix* that we will use is the *rolling rule* (Backhouse, 2002), which allows the first argument of a composition to be pulled outside a *fix*, resulting in the composition swapping the order of its arguments, or 'rolling over':

Lemma 2.5 (*Rolling Rule*)

$$\mathit{fix}\,(f \circ g) = f\,(\mathit{fix}\,(g \circ f))$$

Proof
See Gill & Hutton (2009). □

3 Worker/Wrapper for Least Fixed Points

Within the domain-theoretic setting of the previous section, consider a recursive program defined as the least fixed point of a function $f : A \to A$ on some type A. Now consider a more efficient program that performs the same task, defined by first taking the least fixed point of a function $g : B \to B$ on some other type B, and then converting the resulting value back to the original type by applying a function $\mathit{abs} : B \to A$. The equivalence between these two programs is captured by the following equation:

$$\mathit{fix}\,f = \mathit{abs}\,(\mathit{fix}\,g)$$

We call $\mathit{fix}\,f$ the original program, $\mathit{fix}\,g$ the *worker* program, *abs* the *wrapper* function, and the equation itself the *worker/wrapper factorisation* for least fixed points. We now turn our attention to identifying conditions to ensure that it holds.

3.1 Assumptions and Conditions

First, we require an additional conversion function $\mathit{rep} : A \to B$ from the original type to the new type. This function is not required to be an inverse of *abs*, but we do require one of the following *worker/wrapper assumptions* to hold:

(A) $\mathit{abs} \circ \mathit{rep} = \mathit{id}_A$
(B) $\mathit{abs} \circ \mathit{rep} \circ f = f$
(C) $\mathit{fix}\,(\mathit{abs} \circ \mathit{rep} \circ f) = \mathit{fix}\,f$

These assumptions form a hierarchy, with (A) ⇒ (B) ⇒ (C). Assumption (A) is the strongest and usually the easiest to verify, and states that *abs* is a left inverse of *rep*, which in the terminology of data refinement means that the *abstract* type A can be faithfully *represented* by the concrete type B. For some applications, however, assumption (A) may not be true in general, but only for values produced by the body function f of the original program, as captured by the weaker assumption (B), or we may also need to take the recursive context into account, as captured by (C).

Additionally, we require one of the following *worker/wrapper conditions*[3] that relate the body functions f and g of the original and worker programs:

[3] The assumptions and conditions are both sets of equational properties; we use the differing terminology for consistency with Gill & Hutton (2009) and Hutton *et al.* (2010).

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

(2) $rep \circ f = g \circ rep \wedge$ strict rep (2β) $fix\ g = rep\ (fix\ f)$

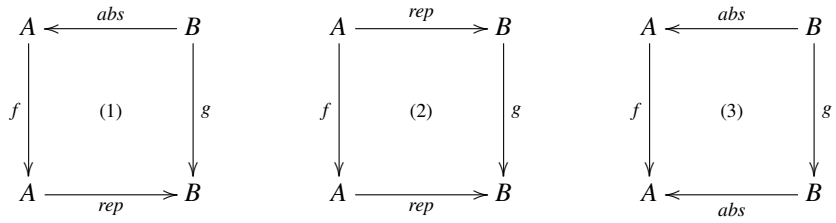
(3) $abs \circ g = f \circ abs$

In general, there is no relationship between the conditions in the first column, i.e. none implies any of the others, while the β conditions in the second column arise as weaker versions of the corresponding conditions in the first. The implications (1) $\Rightarrow$ (1β) and (2) $\Rightarrow$ (2β) follow immediately using extensionality and fixed-point fusion respectively, which in the latter case accounts for the strictness side condition in (2). We will return to the issue of strictness in Section 3.2. Furthermore, given assumption (C), it is straightforward to show that conditions (1β) and (2β) are in fact equivalent:

$$
\begin{array}{ll}
 & fix\ g = rep\ (fix\ f) \\
\Leftrightarrow & \quad \{\ \text{computation}\ \} \\
 & fix\ g = rep\ (f\ (fix\ f)) \\
\Leftrightarrow & \quad \{\ \text{(C)}\ \} \\
 & fix\ g = rep\ (f\ (fix\ (abs \circ rep \circ f))) \\
\Leftrightarrow & \quad \{\ \text{rolling rule}\ \} \\
 & fix\ g = fix\ (rep \circ f \circ abs)
\end{array}
$$

Nonetheless, it is still useful to consider both conditions, as in some situations one may be simpler to use than the other. Note that attempting to weaken condition (3) in a similar manner gives $fix\ f = abs\ (fix\ g)$, which there is no merit in considering as this is precisely the worker/wrapper factorisation result that we wish to establish.

The stronger conditions can also be expressed as commuting diagrams, in which case strictness of *rep* then becomes a side condition on the second diagram:



In terms of how the worker/wrapper conditions are used in practice, for some applications the worker program *fix g* will already be given, and our aim then is to *verify* that one of the conditions is satisfied. In such cases, we use the condition that admits the simplest verification, which is often one of the stronger conditions (1), (2) or (3) that do not involve the use of *fix*. For other applications, our aim will be to *construct* the worker program. In such cases, conditions (1), (1β) or (2β) provide explicit but inefficient definitions for the worker program in terms of the body function f of the original program, which we then attempt to make more efficient using program-fusion techniques. This was the approach that was taken by Gill & Hutton (2009). However, as shown by Hutton *et al.* (2010), in some cases it is preferable to use conditions (2) or (3), which provide an indirect *specification* for the body function g of the worker, rather than a direct definition.

3.2 Worker/Wrapper Factorisation

We can now state the main result of this section: provided that any of the worker/wrapper assumptions hold, and any of the worker/wrapper conditions hold, then worker/wrapper factorisation is valid, as summarised in Figure 1. To prove this result it suffices to consider assumption (C) and conditions (1β) and (3) in turn, as (A), (B), (1) and (2) are already covered by their weaker versions, and (2β) is equivalent to (1β) in the presence of (C).

Proof
For condition (1β):

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

□

Proof
For condition (3), at first glance it may appear that we don't need assumption (C) at all, as condition (3) on its own is sufficient by fusion:

$$\begin{array}{ll} & \mathit{abs}\, (\mathit{fix}\, g) = \mathit{fix}\, f \\ \Leftarrow & \quad \{\ \text{fusion}\ \} \\ & \mathit{abs} \circ g = f \circ \mathit{abs} \wedge \text{strict}\, \mathit{abs} \\ \Leftrightarrow & \quad \{\ \text{(3)}\ \} \\ & \text{strict}\, \mathit{abs} \end{array}$$

But this proof requires that *abs* is strict. However, using assumption (C) we can in fact prove the factorisation result without this extra strictness condition:

$$\begin{array}{ll} & \mathit{fix}\, f = \mathit{abs}\, (\mathit{fix}\, g) \\ \Leftarrow & \quad \{\ \text{antisymmetry}\ \} \\ & \mathit{fix}\, f \sqsubseteq \mathit{abs}\, (\mathit{fix}\, g) \wedge \mathit{abs}\, (\mathit{fix}\, g) \sqsubseteq \mathit{fix}\, f \end{array}$$

We now verify the two inclusions separately:

$$\begin{array}{ll} & \mathit{fix}\, f \sqsubseteq \mathit{abs}\, (\mathit{fix}\, g) \\ \Leftarrow & \quad \{\ \text{induction}\ \} \\ & f\, (\mathit{abs}\, (\mathit{fix}\, g)) \sqsubseteq \mathit{abs}\, (\mathit{fix}\, g) \\ \Leftrightarrow & \quad \{\ \text{(3)}\ \} \\ & \mathit{abs}\, (g\, (\mathit{fix}\, g)) \sqsubseteq \mathit{abs}\, (\mathit{fix}\, g) \\ \Leftrightarrow & \quad \{\ \text{computation}\ \} \\ & \mathit{abs}\, (\mathit{fix}\, g) \sqsubseteq \mathit{abs}\, (\mathit{fix}\, g) \\ \Leftrightarrow & \quad \{\ \text{reflexivity}\ \} \\ & \mathit{True} \end{array}$$

and

$$
\begin{array}{ll}
 & abs\ (fix\ g) \sqsubseteq fix\ f \\
\Leftrightarrow & \quad \{ \text{ define } P\ y = abs\ y \sqsubseteq fix\ f \ \} \\
 & P\ (fix\ g) \\
\Leftarrow & \quad \{ \text{ fixed-point induction } \} \\
 & P \perp \wedge \forall y.\ P\ y \Rightarrow P\ (g\ y)
\end{array}
$$

We now verify the two conjuncts separately:

$$
\begin{array}{ll}
 & P \perp \\
\Leftrightarrow & \quad \{ \text{ applying } P \ \} \\
 & abs \perp \sqsubseteq fix\ f \\
\Leftrightarrow & \quad \{ \text{ (C) } \} \\
 & abs \perp \sqsubseteq fix\ (abs \circ rep \circ f) \\
\Leftrightarrow & \quad \{ \text{ rolling rule } \} \\
 & abs \perp \sqsubseteq abs\ (fix\ (rep \circ f \circ abs)) \\
\Leftarrow & \quad \{ \text{ monotonicity } \} \\
 & \perp \sqsubseteq fix\ (rep \circ f \circ abs) \\
\Leftrightarrow & \quad \{ \text{ bottom } \} \\
 & True
\end{array}
$$

and

$$
\begin{array}{ll}
 & P\ (g\ y) \\
\Leftrightarrow & \quad \{ \text{ applying } P \ \} \\
 & abs\ (g\ y) \sqsubseteq fix\ f \\
\Leftrightarrow & \quad \{ \text{ (3) } \} \\
 & f\ (abs\ y) \sqsubseteq fix\ f \\
\Leftrightarrow & \quad \{ \text{ computation } \} \\
 & f\ (abs\ y) \sqsubseteq f\ (fix\ f) \\
\Leftarrow & \quad \{ \text{ monotonicity } \} \\
 & abs\ y \sqsubseteq fix\ f \\
\Leftrightarrow & \quad \{ \text{ unapplying } P \ \} \\
 & P\ y
\end{array}
$$

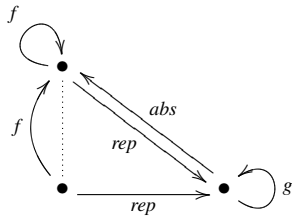
□

Hence, we have seen that given assumption (C) and condition (3), we do not require that *abs* is strict to ensure worker/wrapper factorisation. But perhaps *abs* being strict is implied in this context? In fact, given assumption (A), this is the case:

$$
\begin{array}{ll}
 & abs \perp = \perp \\
\Leftarrow & \quad \{ \text{ antisymmetry } \} \\
 & abs \perp \sqsubseteq \perp \wedge \perp \sqsubseteq abs \perp \\
\Leftrightarrow & \quad \{ \text{ bottom } \} \\
 & abs \perp \sqsubseteq \perp \\
\Leftrightarrow & \quad \{ \text{ identity } \} \\
 & abs \perp \sqsubseteq id_A \perp \\
\Leftrightarrow & \quad \{ \text{ (A) } \}
\end{array}
$$

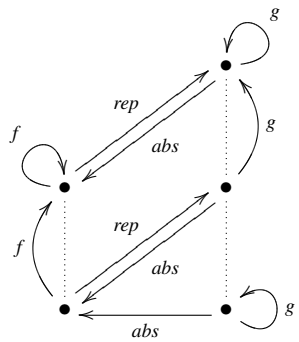
$$
\begin{array}{ll}
 & \mathit{abs}\ \bot \sqsubseteq \mathit{abs}\ (\mathit{rep}\ \bot) \\
\Leftarrow & \qquad \{\ \text{monotonicity}\ \} \\
 & \bot \sqsubseteq \mathit{rep}\ \bot \\
\Leftrightarrow & \qquad \{\ \text{bottom}\ \} \\
 & \mathit{True}
\end{array}
$$

However, for the weaker assumption (B), *abs* is not necessarily strict. A simple counterexample is shown in the following diagram, in which bullets on the left and right sides are elements of A and B respectively, dotted lines are orderings ($x \sqsubseteq y$) that are directed upwards, and solid arrows are mappings ($x \mapsto y$):



In particular, this example satisfies assumption (B), condition (3), and worker/wrapper factorisation, but *abs* is non-strict. Because (B) implies (C), the same counterexample also shows that the strictness of *abs* is not implied by (C) and (3). It is interesting to note that in the past condition (3) was regarded as being uninteresting because it just corresponds to the use of fusion (Hutton *et al.*, 2010). But in the context of *fix* this requires that *abs* is strict. However, as we have now seen, in the case of (B) and (C) this requirement can be dropped. Hence, worker/wrapper factorisation for condition (3) is applicable in some situations where fusion is not, i.e. when *abs* is non-strict.

Recall that showing (2) $\Rightarrow$ (2β) using fixed-point fusion required that *rep* is strict. It is natural to ask if we can drop strictness from (2) by proving worker/wrapper factorisation in another way, as we did above with condition (3). The answer is no, and we verify this by exhibiting a non-strict *rep* that satisfies $\mathit{rep} \circ f = g \circ \mathit{rep}$ and assumption (A), but for which worker/wrapper factorisation does not hold, as follows:



Because (A) $\Rightarrow$ (B) $\Rightarrow$ (C), the same example shows that $\mathit{rep} \circ f = g \circ \mathit{rep}$ on its own is also insufficient for assumptions (B) and (C). However, while the addition of strictness is sufficient to ensure worker/wrapper factorisation, it is not *necessary*, which can be verified by exhibiting a non-strict *rep* that satisfies $\mathit{rep} \circ f = g \circ \mathit{rep}$, assumption (A),

Given functions

$$f : A \to A$$
$$g : B \to B$$

for some types A and B, and conversion functions

$$rep : A \to B$$
$$abs : B \to A$$

then we have a set of *worker/wrapper assumptions*

(A) $abs \circ rep = id_A$
(B) $abs \circ rep \circ f = f$
(C) $fix\,(abs \circ rep \circ f) = fix\,f$

and a set of *worker/wrapper conditions*

(1) $g = rep \circ f \circ abs$
(2) $rep \circ f = g \circ rep \wedge$ strict rep
(3) $abs \circ g = f \circ abs$

(1β) $fix\,g = fix\,(rep \circ f \circ abs)$
(2β) $fix\,g = rep\,(fix\,f)$

Provided that any of the assumptions hold and any of the conditions hold, then *worker/wrapper factorisation* is valid:

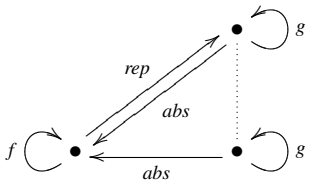
$$fix\,f = abs\,(fix\,g)$$

Furthermore, if any of the assumptions hold, and any of the conditions except (3) hold, then *worker/wrapper fusion* is valid:

$$rep\,(abs\,(fix\,g)) = fix\,g$$

Figure 1: Worker/wrapper transformation for least fixed points.

and worker/wrapper factorisation, shown in the example below. As before, this example also verifies that strictness is not necessary for (B) and (C).



3.3 Worker/Wrapper Fusion

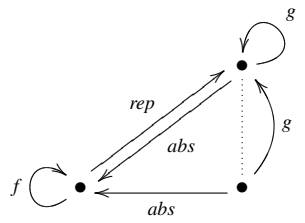
When applying worker/wrapper factorisation, it is often desirable to fuse together instances of the conversion functions *rep* and *abs* to eliminate the overhead of repeatedly converting between the new and original types (Gill & Hutton, 2009). In general, it is not the case that $rep \circ abs$ can be fused to give id_B. However, provided that any of the assumptions (A), (B) or (C) hold, and any of the conditions except (3) hold, then the following *worker/wrapper-fusion* property is valid, as summarised in Figure 1:

$$rep\,(abs\,(fix\,g)) = fix\,g$$

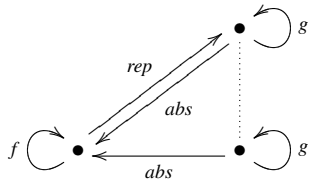
In a similar manner to Section 3.2, for the purposes of proving this result it suffices to consider assumption (C) and condition (2β):

$$\begin{array}{ll} & \mathit{rep}\ (\mathit{abs}\ (\mathit{fix}\ g)) \\ = & \quad \{\ \text{worker/wrapper factorisation, (C) and } (2\beta)\ \} \\ & \mathit{rep}\ (\mathit{fix}\ f) \\ = & \quad \{\ (2\beta)\ \} \\ & \mathit{fix}\ g \end{array}$$

As with worker/wrapper factorisation, we confirm that strictness of *rep* is sufficient but not necessary in the case of condition (2), by exhibiting a non-strict *rep* that satisfies $\mathit{rep} \circ f = g \circ \mathit{rep}$, assumption (A), and worker/wrapper fusion:



Finally, in the case of condition (3), the following example shows that (3) and (A) are not sufficient to ensure worker/wrapper fusion:



Furthermore, even if we were also to require that *rep* be strict, *abs* be strict, or both conversion functions be strict, it is still possible to construct corresponding examples that demonstrate that worker/wrapper fusion does not hold in general for condition (3).

3.4 Relationship to Previous Work

The worker/wrapper results for *fix* presented in this section generalise those in Gill & Hutton (2009). The key difference is that the original article only considered worker/wrapper factorisation for condition (1β), although it wasn't identified as an explicit condition but rather inlined in the statement of the theorem itself, whereas we have shown that the result is also valid for (1), (2), (2β) and (3). Moreover, worker/wrapper fusion was only established for assumption (A) and condition (1β), whereas we have shown that any of the assumptions (A), (B) or (C) and any of the conditions (1), (1β), (2) or (2β) are sufficient. We also exhibited a counterexample to show that (3) is not a sufficient condition for worker/wrapper fusion under any of the assumptions.

We conclude by noting that in the context of assumption (C), the equivalent conditions (1β) and (2β) are not just sufficient to ensure that worker/wrapper factorisation and fusion hold, but are in fact necessary too. In particular, given these two properties, we can then verify that condition (2β) holds by the following simple calculation:

$$
\begin{array}{ll}
 & \mathit{fix}\ g \\
= & \quad \{\ \text{worker/wrapper fusion}\ \} \\
 & \mathit{rep}\ (\mathit{abs}\ (\mathit{fix}\ g)) \\
= & \quad \{\ \text{worker/wrapper factorisation}\ \} \\
 & \mathit{rep}\ (\mathit{fix}\ f)
\end{array}
$$

Hence, while previous work identified conditions that are sufficient to ensure factorisation and fusion are valid, we now have conditions that are both necessary *and* sufficient.

3.5 Example: The Last Function

To illustrate our new worker/wrapper theory for least fixed points, we use a Haskell-like notation for defining continuous functions on cpos (though note that we use :: for constructing lists). For our first example, we show how optimising a function that returns the last element of a list, the motivating example from Peyton Jones (2007), can be expressed as an instance of the worker/wrapper transformation. We start from the following recursive definition, which in the case of the empty list returns $\bot$, the semantic representation of terminating with an error message:

$$
\begin{array}{lll}
\mathit{last} & : & [A] \to A \\
\mathit{last}\ [\,] & = & \bot \\
\mathit{last}\ [x] & = & x \\
\mathit{last}\ (x :: \mathit{xs}) & = & \mathit{last}\ \mathit{xs}
\end{array}
$$

While correct, this definition is inefficient, because it checks if the argument list is empty at every recursive call of the function, whereas it is clear that once we reach the recursive call the argument is guaranteed to be non-empty. We now show how to eliminate this inefficiency using the worker/wrapper transformation.

The first step is to define *last* as the least fixed point of a non-recursive function, by abstracting over the recursive call in the body of its definition:

$$
\begin{array}{lll}
\mathit{last} & : & [A] \to A \\
\mathit{last} & = & \mathit{fix}\ f \\
f & : & ([A] \to A) \to ([A] \to A) \\
f\ h\ [\,] & = & \bot \\
f\ h\ [x] & = & x \\
f\ h\ (x :: \mathit{xs}) & = & h\ \mathit{xs}
\end{array}
$$

In order to avoid repeatedly checking the empty-list case, we now seek to transform the original function of type $[A] \to A$ into a worker function of type $A \to [A] \to A$, whose first and second arguments are the head and tail of a list, and thereby only applicable to non-empty lists. Defining the necessary conversion functions between the two types is simply a matter of constructing and deconstructing a list:

$$
\begin{array}{lll}
\mathit{rep} & : & ([A] \to A) \to (A \to [A] \to A) \\
\mathit{rep}\ h\ x\ \mathit{xs} & = & h\ (x :: \mathit{xs})
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{abs} & : & (A \to [A] \to A) \to ([A] \to A) \\
\mathit{abs}\ i\ [\,] & = & \bot \\
\mathit{abs}\ i\ (x :: \mathit{xs}) & = & i\ x\ \mathit{xs}
\end{array}
$$

We now need to show that one of the worker/wrapper assumptions holds. While the strongest assumption (A) holds for this example, its proof relies on the fact that polymorphic functions of type $[A] \to A$ must map the empty list to $\bot$, which requires techniques beyond simple calculation to prove (Wadler, 1989). At this point we reiterate that the purpose of introducing several assumptions was to make it as easy as possible to apply worker/wrapper factorisation, by allowing us to choose whichever assumption is simplest to verify for the application at hand. For this example, it turns out to be easier to verify assumption (B), i.e. $\mathit{abs} \circ \mathit{rep} \circ f = f$, which expands to

$$(\mathit{abs} \circ \mathit{rep} \circ f)\ h\ \mathit{xs} = f\ h\ \mathit{xs}$$

and can then be verified as follows:

$$
\begin{array}{ll}
 & (\mathit{abs} \circ \mathit{rep} \circ f)\ h\ \mathit{xs} \\
= & \quad \{ \text{ applying } \circ \ \} \\
 & \mathit{abs}\ (\mathit{rep}\ (f\ h))\ \mathit{xs} \\
= & \quad \{ \text{ applying } \mathit{abs} \ \} \\
 & \textbf{case}\ \mathit{xs}\ \textbf{of} \\
 & \quad [\,] \to \bot \\
 & \quad (y :: \mathit{ys}) \to \mathit{rep}\ (f\ h)\ y\ \mathit{ys} \\
= & \quad \{ \text{ applying } \mathit{rep} \ \} \\
 & \textbf{case}\ \mathit{xs}\ \textbf{of} \\
 & \quad [\,] \to \bot \\
 & \quad (y :: \mathit{ys}) \to f\ h\ (y :: \mathit{ys}) \\
= & \quad \{ \text{ applying } f \ \} \\
 & \textbf{case}\ \mathit{xs}\ \textbf{of} \\
 & \quad [\,] \to \bot \\
 & \quad (y :: \mathit{ys}) \to \textbf{case}\ \mathit{ys}\ \textbf{of} \\
 & \qquad\qquad [\,] \to y \\
 & \qquad\qquad \mathit{ys} \to h\ \mathit{ys} \\
= & \quad \{ \text{ combining } \textbf{case} \text{ expressions } \} \\
 & \textbf{case}\ \mathit{xs}\ \textbf{of} \\
 & \quad [\,] \to \bot \\
 & \quad [y] \to y \\
 & \quad (y :: \mathit{ys}) \to h\ \mathit{ys} \\
= & \quad \{ \text{ unapplying } f \ \} \\
 & f\ h\ \mathit{xs}
\end{array}
$$

The most convenient worker/wrapper condition to use as the basis for constructing the worker function is condition (1), i.e. $g = \mathit{rep} \circ f \circ \mathit{abs}$, which expands to

$$g\ i\ x\ \mathit{xs} = \mathit{rep}\ (f\ (\mathit{abs}\ i))\ x\ \mathit{xs}$$

and can be then be rewritten as follows:

$$
\begin{array}{ll}
 & g\ i\ x\ xs \\
= & \qquad \{ \text{ above equation } \} \\
 & rep\ (f\ (abs\ i))\ x\ xs \\
= & \qquad \{ \text{ applying } rep \} \\
 & f\ (abs\ i)\ (x :: xs) \\
= & \qquad \{ \text{ applying } f \} \\
 & \textbf{case}\ xs\ \textbf{of} \\
 & \quad [\,] \quad \to x \\
 & \quad (y :: ys) \to abs\ i\ (y :: ys) \\
= & \qquad \{ \text{ applying } abs \} \\
 & \textbf{case}\ xs\ \textbf{of} \\
 & \quad [\,] \quad \to x \\
 & \quad (y :: ys) \to i\ y\ ys
\end{array}
$$

That is, we have calculated the following definition:

$$
\begin{array}{lll}
g & : & (A \to [A] \to A) \to (A \to [A] \to A) \\
g\ i\ x\ [\,] & = & x \\
g\ i\ x\ (y :: ys) & = & i\ y\ ys
\end{array}
$$

Now that we have satisfied the necessary preconditions, applying worker/wrapper factorisation gives the following new definitions:

$$
\begin{array}{lll}
last & : & [A] \to A \\
last & = & abs\ work \\
work & : & A \to [A] \to A \\
work & = & fix\ g
\end{array}
$$

Finally, if we make the arguments explicit, expand the definition of *abs*, and rewrite the worker function using explicit recursion, we obtain a more efficient version of *last* that only checks once if the initial list is empty:

$$
\begin{array}{lll}
last & : & [A] \to A \\
last\ [\,] & = & \bot \\
last\ (x :: xs) & = & work\ x\ xs \\
work & : & A \to [A] \to A \\
work\ x\ [\,] & = & x \\
work\ x\ (y :: ys) & = & work\ y\ ys
\end{array}
$$

We conclude with some observations about the above derivation. First, note that once we made the design decision to represent a non-empty list using separate head and tail components, the rest of the derivation proceeds in a straightforward manner using simple equational reasoning. Secondly, the derivation does not require the use of induction, as the necessary inductive reasoning is encapsulated in the underlying worker/wrapper theory. And finally, the more efficient version of *last* can of course also be derived using more elementary techniques, but as we have seen it fits naturally into the general worker/wrapper approach to improving the performance of programs by changing their types.

3.6 Example: The Fibonacci Function

For our second example, we derive an efficient version of the Fibonacci function, which shows how the original derivation in Burstall & Darlington (1977) can be viewed as an instance of the worker/wrapper transformation. We begin with the standard recursive definition for the *fib* function on the natural numbers:

$$\begin{array}{ll} \mathit{fib} & : \mathbb{N} \to \mathbb{N} \\ \mathit{fib}\,0 & = 0 \\ \mathit{fib}\,1 & = 1 \\ \mathit{fib}\,(n+2) & = \mathit{fib}\,n + \mathit{fib}\,(n+1) \end{array}$$

This definition directly encodes the intended behaviour of the function, but is inefficient because it recomputes the same results many times over, in fact requiring exponential time in the value of its argument. We now show how it can be transformed into an equivalent but more efficient definition. As previously, the first step is to redefine the function in the form $\mathit{fib} = \mathit{fix}\,f$ for some non-recursive function f, but for this example it turns out that we won't actually need to use the definition for f, so this is omitted.

In order to avoid recomputing results, we note that each successive Fibonacci number is given by the sum of the previous two. Based upon this observation, we now seek to transform the original function of type $\mathbb{N} \to \mathbb{N}$ into a worker function of type $\mathbb{N} \to (\mathbb{N},\mathbb{N})$ that returns two successive results rather than a single result, thereby opening up the possibility of these results being reused to compute the next result. The necessary conversion functions between the two types are defined as follows:

$$\begin{array}{ll} \mathit{rep} & : (\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to (\mathbb{N},\mathbb{N})) \\ \mathit{rep}\,h\,n & = (h\,n,\ h\,(n+1)) \\ \mathit{abs} & : (\mathbb{N} \to (\mathbb{N},\mathbb{N})) \to (\mathbb{N} \to \mathbb{N}) \\ \mathit{abs}\,i\,n & = \mathit{fst}\,(i\,n) \end{array}$$

It is easy to show that assumption (A), i.e. $\mathit{abs} \circ \mathit{rep} = \mathit{id}_{\mathbb{N}\to\mathbb{N}}$, now holds:

$$\begin{array}{ll} & (\mathit{abs} \circ \mathit{rep})\,h\,n \\ = & \quad \{ \text{ applying } \circ \text{ } \} \\ & \mathit{abs}\,(\mathit{rep}\,h)\,n \\ = & \quad \{ \text{ applying } \mathit{abs} \text{ } \} \\ & \mathit{fst}\,(\mathit{rep}\,h\,n) \\ = & \quad \{ \text{ applying } \mathit{rep} \text{ } \} \\ & \mathit{fst}\,(h\,n, h\,(n+1)) \\ = & \quad \{ \text{ applying } \mathit{fst} \text{ } \} \\ & h\,n \end{array}$$

In order to construct the worker function $\mathit{work} = \mathit{fix}\,g$, the appropriate condition to use as the starting point for this example is condition (2β), i.e. $\mathit{fix}\,g = \mathit{rep}\,(\mathit{fix}\,f)$, or equivalently $\mathit{fix}\,g\,n = \mathit{rep}\,(\mathit{fix}\,f)\,n$, which can then be rewritten as follows:

$$\begin{array}{ll} & \mathit{fix}\,g\,n = \mathit{rep}\,(\mathit{fix}\,f)\,n \\ \Leftrightarrow & \quad \{ \text{ unapplying } \mathit{work} \text{ and } \mathit{fib} \text{ } \} \\ & \mathit{work}\,n = \mathit{rep}\,\mathit{fib}\,n \end{array}$$

$$\begin{array}{ll} \Leftrightarrow & \{ \text{ applying } \mathit{rep} \ \} \\ & \mathit{work}\ n\ =\ (\mathit{fib}\ n, \mathit{fib}\ (n+1)) \end{array}$$

Hence, for condition (2β) to be valid we require a function *work* that satisfies the equation $\mathit{work}\ n = (\mathit{fib}\ n, \mathit{fib}\ (n+1))$. Using this as a specification of the desired behaviour, we calculate an implementation by constructive induction on the natural number n. For the base cases $n = \bot$ and $n = 0$, expanding the above equation gives $\mathit{work}\ \bot = (\bot, \bot)$ and $\mathit{work}\ 0 = (0,1)$, while for the inductive case we calculate as follows:

$$\begin{array}{ll} & \mathit{work}\ (n+1) \\ = & \quad \{ \text{ above equation } \} \\ & (\mathit{fib}\ (n+1), \mathit{fib}\ (n+2)) \\ = & \quad \{ \text{ applying } \mathit{fib} \ \} \\ & (\mathit{fib}\ (n+1), \mathit{fib}\ n + \mathit{fib}\ (n+1)) \\ = & \quad \{ \textbf{let} \text{ abstraction } \} \\ & \textbf{let}\ (x,y)\ =\ (\mathit{fib}\ n, \mathit{fib}\ (n+1))\ \textbf{in}\ (y, x+y) \\ = & \quad \{ \text{ induction hypothesis } \} \\ & \textbf{let}\ (x,y)\ =\ \mathit{work}\ n\ \textbf{in}\ (y, x+y) \end{array}$$

In summary, we have calculated the following definition (as usual, we omit the $\bot$ case as this is accounted for by the semantics of pattern matching):

$$\begin{array}{lll} \mathit{work} & : & \mathbb{N} \to (\mathbb{N}, \mathbb{N}) \\ \mathit{work}\ 0 & = & (0,1) \\ \mathit{work}\ (n+1) & = & \textbf{let}\ (x,y)\ =\ \mathit{work}\ n\ \textbf{in}\ (y, x+y) \end{array}$$

Finally, applying worker/wrapper factorisation gives $\mathit{fib} = \mathit{abs}\ \mathit{work}$, which by making the argument explicit and expanding the definition of the conversion function *abs* results in a linear time version of the Fibonacci function:

$$\begin{array}{lll} \mathit{fib} & : & \mathbb{N} \to \mathbb{N} \\ \mathit{fib}\ n & = & \mathit{fst}\ (\mathit{work}\ n) \end{array}$$

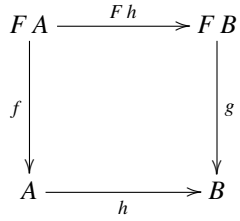
As with the *last* example, note that once we made the initial decision regarding the change in type of the *fib* function, applying the worker/wrapper transformation amounts to routine equational reasoning, which in this case required a simple inductive calculation. Note also that this derivation involved changing the result type of a function, whereas the previous example changed the argument type.

4 Initial-Algebra Semantics

We now turn our attention to the other previous formalisation of the worker/wrapper transformation, which was based upon an initial-algebra semantics of recursion in a categorical setting in which programs are defined using fold operators. In this section we review the basic definitions and properties from this approach to program semantics, and introduce our notation. For further details, see for example Bird & de Moor (1997).

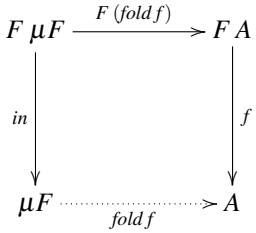
4.1 Basic Definitions

Suppose that we fix a category $\mathbb{C}$ and a functor $F : \mathbb{C} \to \mathbb{C}$ on this category. Then an *F-algebra* is a pair (A,f) comprising an object A and an arrow $f : F\ A \to A$. An F-homomorphism from one such algebra (A,f) to another (B,g) is an arrow $h : A \to B$ such that the following diagram commutes:



4.2 Initial Algebras

Algebras and homomorphisms themselves form a category, with composition and identities inherited from the original category $\mathbb{C}$. An *initial algebra* is an initial object in this new category, and we write $(\mu F, in)$ for an initial F-algebra, and $fold\ f$ for the unique homomorphism from this initial algebra to any other algebra (A,f). That is, $fold\ f$ is the unique arrow that makes the following diagram commute:



Moreover, the arrow $in : F\ \mu F \to \mu F$ has an inverse $out : \mu F \to F\ \mu F$, which establishes an isomorphism $F\ \mu F \cong \mu F$. The above definition for $fold\ f$ can also be expressed as the following equivalence, known as the *universal property* of *fold*:

Lemma 4.1 (*Universal Property of Fold*)

$$h = fold\ f \ \Leftrightarrow\ h \circ in = f \circ F\ h$$

The $\Rightarrow$ direction of this equivalence states that $fold\ f$ is a homomorphism from the initial algebra $(\mu F, in)$ to another algebra (A,f), while the $\Leftarrow$ direction states that any other such homomorphism h must be equal to $fold\ f$. The universal property forms the basic proof technique for the fold operator.

4.3 Useful Properties

In a similar manner to fixed-point fusion (Lemma 2.4), we also have a fusion property for folds, which states that the composition of a homomorphism of the appropriate type and a fold can always be re-expressed as a single fold:

Lemma 4.2 (*Fold Fusion*)

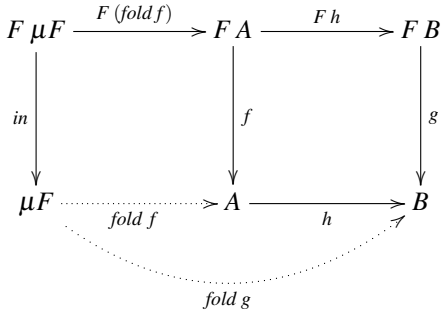
$$h \circ f = g \circ F\,h \;\Rightarrow\; h \circ \mathit{fold}\,f = \mathit{fold}\,g$$

Proof

$$\begin{array}{ll} & h \circ \mathit{fold}\,f = \mathit{fold}\,g \\ \Leftrightarrow & \quad \{\ \text{universal property}\ \} \\ & h \circ \mathit{fold}\,f \circ \mathit{in} = g \circ F\,(h \circ \mathit{fold}\,f) \\ \Leftrightarrow & \quad \{\ \text{functors}\ \} \\ & h \circ \mathit{fold}\,f \circ \mathit{in} = g \circ F\,h \circ F\,(\mathit{fold}\,f) \\ \Leftrightarrow & \quad \{\ \mathit{fold}\,f \text{ is a homomorphism}\ \} \\ & h \circ f \circ F\,(\mathit{fold}\,f) = g \circ F\,h \circ F\,(\mathit{fold}\,f) \\ \Leftarrow & \quad \{\ \text{extensionality}\ \} \\ & h \circ f = g \circ F\,h \end{array}$$

□

It is also useful to express fold fusion in the form of a commuting diagram as shown below, in which commutativity of the left square is given by the universal property of fold, commutativity of the right square is given by the fusion precondition, and commutativity of the lower component, i.e. $h \circ \mathit{fold}\,f = \mathit{fold}\,g$, then follows by the above calculation:



The rolling rule (Lemma 2.5) can also be formulated for folds, and is sometimes useful for rewriting proofs using fusion in a purely equational manner:

Lemma 4.3 (*Rolling Rule*)

$$\mathit{fold}\,(f \circ g) = f \circ \mathit{fold}\,(g \circ F\,f)$$

Proof

$$\begin{array}{ll} & \mathit{fold}\,(f \circ g) = f \circ \mathit{fold}\,(g \circ F\,f) \\ \Leftarrow & \quad \{\ \text{fold fusion}\ \} \\ & f \circ g \circ F\,f = f \circ g \circ F\,f \\ \Leftrightarrow & \quad \{\ \text{reflexivity}\ \} \\ & \mathit{True} \end{array}$$

□

In turn, the special case of the rolling rule when g is the identity function provides a variant of the computation rule (Lemma 2.1) for folds:

Lemma 4.4 (*Computation*)

$$\mathit{fold}\, f \;=\; f \circ \mathit{fold}\, (F\, f)$$

Proof

$$\begin{array}{ll} & \mathit{fold}\, f \\ = & \quad \{ \text{ identities } \} \\ & \mathit{fold}\, (f \circ \mathit{id}) \\ = & \quad \{ \text{ rolling rule } \} \\ & f \circ \mathit{fold}\, (\mathit{id} \circ F\, f) \\ = & \quad \{ \text{ identities } \} \\ & f \circ \mathit{fold}\, (F\, f) \end{array}$$

□

While the universal property provides a unique characterisation of the fold operator, it can sometimes be useful to have an explicit recursive definition:

Lemma 4.5 (*Definition of Fold*)

$$\mathit{fold}\, f \;=\; f \circ F\, (\mathit{fold}\, f) \circ \mathit{out}$$

Proof

$$\begin{array}{ll} & \mathit{fold}\, f \;=\; f \circ F\, (\mathit{fold}\, f) \circ \mathit{out} \\ \Leftrightarrow & \quad \{\; \mathit{in} \circ \mathit{out} \;=\; \mathit{id} \;\} \\ & \mathit{fold}\, f \circ \mathit{in} \circ \mathit{out} \;=\; f \circ F\, (\mathit{fold}\, f) \circ \mathit{out} \\ \Leftarrow & \quad \{ \text{ extensionality } \} \\ & \mathit{fold}\, f \circ \mathit{in} \;=\; f \circ F\, (\mathit{fold}\, f) \\ \Leftarrow & \quad \{ \text{ universal property } \} \\ & \mathit{fold}\, f \;=\; \mathit{fold}\, f \\ \Leftrightarrow & \quad \{ \text{ reflexivity } \} \\ & \mathit{True} \end{array}$$

□

Finally, the *functor property* of *fold* states that if two folds are equal, then the folds that result from applying the underlying functor to each algebra are also equal:

Lemma 4.6 (*Functor Property of Fold*)

$$\mathit{fold}\, f \;=\; \mathit{fold}\, g \;\Rightarrow\; \mathit{fold}\, (F\, f) \;=\; \mathit{fold}\, (F\; g)$$

Proof

$$\begin{array}{ll} & \mathit{fold}\, (F\, f) \\ = & \quad \{ \text{ definition of } \mathit{fold} \} \\ & F\, f \circ F\, (\mathit{fold}\, (F\, f)) \circ \mathit{out} \end{array}$$

$$
\begin{aligned}
&= && \{ \text{ functors } \} \\
&F\,(f \circ \mathit{fold}\,(F\,f)) \circ \mathit{out} \\
&= && \{ \text{ computation } \} \\
&F\,(\mathit{fold}\,f) \circ \mathit{out} \\
&= && \{\, \mathit{fold}\,f \;=\; \mathit{fold}\,g \,\} \\
&F\,(\mathit{fold}\,g) \circ \mathit{out} \\
&= && \{ \text{ reversing above steps } \} \\
&\mathit{fold}\,(F\,g)
\end{aligned}
$$

□

5 Worker/Wrapper for Initial Algebras

Within the category-theoretic setting of the previous section, consider a recursive program defined as the fold of an algebra $f : F\,A \to A$ for some object A. Now consider a more efficient program that performs the same task, defined by first folding an algebra $g : F\,B \to B$ on some other object B, and then converting the resulting value back to the original object type by composing with an arrow $\mathit{abs} : B \to A$. The equivalence between these two programs is captured by the following equation:

$$\mathit{fold}\,f \;=\; \mathit{abs} \circ \mathit{fold}\,g$$

In a similar manner to least fixed points, we call $\mathit{fold}\,f$ the original program, $\mathit{fold}\,g$ the *worker* program, *abs* the *wrapper* arrow, and the equation itself the *worker/wrapper factorisation* for initial algebras. We now seek conditions under which this equation holds.

5.1 Assumptions and Conditions

The conditions that we use to validate worker/wrapper factorisation for initial algebras are similar to those that we identified for least fixed points. In particular, we require a conversion arrow $\mathit{rep} : A \to B$, one of the following assumptions

(A) $\mathit{abs} \circ \mathit{rep} \;=\; \mathit{id}_A$

(B) $\mathit{abs} \circ \mathit{rep} \circ f \;=\; f$

(C) $\mathit{fold}\,(\mathit{abs} \circ \mathit{rep} \circ f) \;=\; \mathit{fold}\,f$

and one of the following conditions:

(1) $g \;=\; \mathit{rep} \circ f \circ F\,\mathit{abs}$

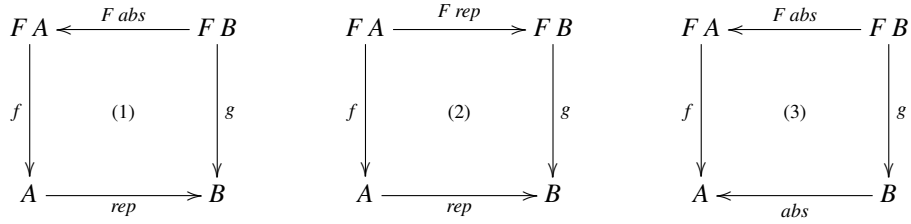
(2) $\mathit{rep} \circ f \;=\; g \circ F\,\mathit{rep}$

(3) $\mathit{abs} \circ g \;=\; f \circ F\,\mathit{abs}$

(1β) $\mathit{fold}\,g \;=\; \mathit{fold}\,(\mathit{rep} \circ f \circ F\,\mathit{abs})$

(2β) $\mathit{fold}\,g \;=\; \mathit{rep} \circ \mathit{fold}\,f$

As with least fixed points, the assumptions form a hierarchy (A) $\Rightarrow$ (B) $\Rightarrow$ (C) and the β conditions are weaker versions of the corresponding conditions in the first column, but in general there is no relationship between conditions (1), (2) and (3). These stronger conditions can also be expressed as commuting diagrams:



In particular, these diagrams make clear that condition (2) states that *rep* is a homomorphism from f to g, and (3) states that *abs* is a homomorphism from g to f. As previously, (1) provides an explicit definition for g in terms of f. As we are working in an arbitrary category the notion of strictness is not defined, and hence there is no requirement that *rep* be strict for (2); we will return to this point in Section 6.

Finally, we note that as with least fixed points the implications (1) $\Rightarrow$ (1β) and (2) $\Rightarrow$ (2β) follow immediately using extensionality and fold fusion respectively, and given assumption (C), conditions (1β) and (2β) are equivalent:

$$
\begin{array}{ll}
 & \mathit{fold}\ g\ =\ \mathit{rep} \circ \mathit{fold}\ f \\
\Leftrightarrow & \qquad \{\ \text{computation}\ \} \\
 & \mathit{fold}\ g\ =\ \mathit{rep} \circ f \circ \mathit{fold}\ (F\ f) \\
\Leftrightarrow & \qquad \{\ \text{(C), functor property}\ \} \\
 & \mathit{fold}\ g\ =\ \mathit{rep} \circ f \circ \mathit{fold}\ (F\ (\mathit{abs} \circ \mathit{rep} \circ f)) \\
\Leftrightarrow & \qquad \{\ \text{functors}\ \} \\
 & \mathit{fold}\ g\ =\ \mathit{rep} \circ f \circ \mathit{fold}\ (F\ \mathit{abs} \circ F\ (\mathit{rep} \circ f)) \\
\Leftrightarrow & \qquad \{\ \text{rolling rule}\ \} \\
 & \mathit{fold}\ g\ =\ \mathit{fold}\ (\mathit{rep} \circ f \circ F\ \mathit{abs})
\end{array}
$$

5.2 Worker/Wrapper Factorisation

We can now state the main result of this section: provided that any of the worker/wrapper assumptions hold, and any of the worker/wrapper conditions hold, then worker/wrapper factorisation is valid. This is the same result as for least fixed points in Section 3, but now in the context of initial algebras, and is summarised in Figure 2. To prove this result we consider assumption (C) and conditions (1β) and (3) in turn, as (A), (B), (1) and (2) are covered by their weaker versions, and (2β) is equivalent to (1β) given (C). The proofs are essentially the same as those for least fixed points.

Proof
For condition (1β):

$$
\begin{array}{ll}
 & \mathit{fold}\ f \\
= & \qquad \{\ \text{(C)}\ \} \\
 & \mathit{fold}\ (\mathit{abs} \circ \mathit{rep} \circ f) \\
= & \qquad \{\ \text{rolling rule}\ \} \\
 & \mathit{abs} \circ \mathit{fold}\ (\mathit{rep} \circ f \circ F\ \mathit{abs}) \\
= & \qquad \{\ (1\beta)\ \} \\
 & \mathit{abs} \circ \mathit{fold}\ g
\end{array}
$$

□

Proof
For condition (3), we don't need assumption (C) at all, as condition (3) on its own is sufficient by a simple application of fusion:

$$
\begin{array}{ll}
 & abs \circ fold\ g\ =\ fold\ f \\
\Leftarrow & \quad \{\ \text{fusion}\ \} \\
 & abs \circ g\ =\ f \circ F\ abs \\
\Leftrightarrow & \quad \{\ (3)\ \} \\
 & True
\end{array}
$$

□

Note that the corresponding proof for least fixed points required that *abs* is strict, and eliminating this extra condition required a much more complicated proof. In contrast, here there is no strictness condition, and the simple proof above suffices.

5.3 Worker/Wrapper Fusion

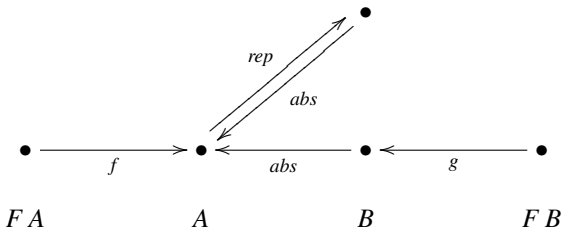
Worker/wrapper-fusion can also be formulated for initial algebras, and holds given any of the assumptions and any of the conditions except (3), as summarised in Figure 2:

$$rep \circ abs \circ fold\ g\ =\ fold\ g$$

We prove this result for the weakest assumption (C) and condition (2β), in a similar manner to the case for least fixed points:

$$
\begin{array}{ll}
 & rep \circ abs \circ fold\ g \\
= & \quad \{\ \text{worker/wrapper factorisation, (C) and } (2\beta)\ \} \\
 & rep \circ fold\ f \\
= & \quad \{\ (2\beta)\ \} \\
 & fold\ g
\end{array}
$$

Finally, we note that that worker/wrapper fusion is not in general valid for condition (3). The counterexample for least fixed points in Section 3.3 can readily be adapted to initial algebras. Specifically, if we define a constant functor $F : \mathbf{SET} \to \mathbf{SET}$ on the category of sets and total functions by $F\ X = \mathbf{1}$ and $F\ f = id_{\mathbf{1}}$, where $\mathbf{1}$ is any singleton set, then the following definitions satisfy (3) and (A) but not worker/wrapper fusion:



Given algebras

$$f : F\,A \to A$$
$$g : F\,B \to B$$

for some functor F, and conversion arrows

$$rep : A \to B$$
$$abs : B \to A$$

then we have a set of *worker/wrapper assumptions*

(A) $abs \circ rep = id_A$
(B) $abs \circ rep \circ f = f$
(C) $fold\,(abs \circ rep \circ f) = fold\,f$

and a set of *worker/wrapper conditions*

(1) $g = rep \circ f \circ F\,abs$ $\qquad$ (1β) $fold\,g = fold\,(rep \circ f \circ F\,abs)$
(2) $rep \circ f = g \circ F\,rep$ $\qquad$ (2β) $fold\,g = rep \circ fold\,f$
(3) $abs \circ g = f \circ F\,abs$

Provided that any of the assumptions hold and any of the conditions hold, then *worker/wrapper factorisation* is valid:

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

Furthermore, if any of the assumptions hold, and any of the conditions except (3) hold, then *worker/wrapper fusion* is valid:

$$rep \circ abs \circ fold\,g = fold\,g$$

Figure 2: Worker/wrapper transformation for initial algebras.

5.4 Relationship to Previous Work

The worker/wrapper results for *fold* presented in this section generalise those in Hutton *et al.* (2010). The key difference is that the original article only considered worker/wrapper factorisation for assumption (A) and conditions (1), (2) and (3) (in which context (1) is stronger than the other two conditions), whereas we have shown that the result is also valid for the weaker assumptions (B) and (C) (in which context (1), (2) and (3) are in general unrelated) and the weaker conditions (1β) and (2β). Moreover, worker/wrapper fusion was essentially only established for assumption (A) and condition (1), whereas we have shown that any of the assumptions (A), (B) or (C) and any of the conditions (1), (1β), (2) or (2β) are sufficient. We also showed that (3) is not sufficient for worker/wrapper fusion under any of the assumptions. Finally, we note that as with least fixed points, in the context of assumption (C) the equivalent conditions (1β) and (2β) are both necessary and sufficient to ensure worker/wrapper factorisation and fusion for initial algebras.That is, given these two properties it is then straightforward to show that (2β) holds:

$$\begin{array}{ll} & fold\,g \\ = & \quad \{\ \text{worker/wrapper fusion}\ \} \\ & rep \circ abs \circ fold\,g \end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ worker/wrapper factorisation } \} \\
& \mathit{rep} \circ \mathit{fold}\ f
\end{array}
$$

5.5 Example: The Remove Duplicates Function

To illustrate our new worker/wrapper theory for initial algebras, we use a Haskell-like notation for defining functions in the category **SET**. Our first example concerns the type of lists, for which the *fold* operator can be defined as follows:

$$
\begin{array}{ll}
\mathit{fold} & : (A \to B \to B) \to B \to [A] \to B \\
\mathit{fold}\ f\ v\ [\,] & = v \\
\mathit{fold}\ f\ v\ (x :: xs) & = f\ x\ (\mathit{fold}\ f\ v\ xs)
\end{array}
$$

That is, the function $\mathit{fold}\ f\ v$ processes a list by replacing the empty list $[\,]$ at the end by the value v, and each constructor $(::)$ within the list by the function f. This definition for *fold* on lists is equivalent to the categorical definition, except that it uses two argument functions rather than combining these as a single algebra.

Now let us consider a function *rmdups* that removes duplicate values from a list of integers, defined in terms of the *fold* operator for lists using an auxiliary function *add* that adds a value to the start of a list if it does not already occur in the list, and a function *elem* that decides if a value occurs as an element of a list:

$$
\begin{array}{ll}
\mathit{rmdups} & : [\mathbb{Z}] \to [\mathbb{Z}] \\
\mathit{rmdups} & = \mathit{fold}\ \mathit{add}\ [\,] \\
\mathit{add} & : \mathbb{Z} \to [\mathbb{Z}] \to [\mathbb{Z}] \\
\mathit{add}\ n\ ns & = \textbf{if}\ \mathit{elem}\ n\ ns\ \textbf{then}\ ns\ \textbf{else}\ n :: ns \\
\mathit{elem} & : \mathbb{Z} \to [\mathbb{Z}] \to \mathit{Bool} \\
\mathit{elem}\ n\ [\,] & = \mathit{False} \\
\mathit{elem}\ n\ (m :: ms) & = \textbf{if}\ n == m\ \textbf{then}\ \mathit{True}\ \textbf{else}\ \mathit{elem}\ n\ ms
\end{array}
$$

Notice that this definition of the *rmdups* functions retains the last occurrence of each value in the list; for example, $\mathit{rmdups}\ [1,2,3,2,1] = [3,2,1]$.

The *elem* function takes linear time in the length of its second argument in the case when the value does not occur in the list. Consequently, *rmdups* takes quadratic time. We now show that it can be transformed into a more efficient version that uses an extra data structure to allow the *elem* operation to be performed more efficiently.

Suppose that we are given an abstract type $\mathit{Set}\ \mathbb{Z}$ of sets of integer values, together with the following collection of primitive operations on such sets:

$$
\begin{array}{ll}
\mathit{empty} & : \mathit{Set}\ \mathbb{Z} \\
\mathit{insert} & : \mathbb{Z} \to \mathit{Set}\ \mathbb{Z} \to \mathit{Set}\ \mathbb{Z} \\
\mathit{member} & : \mathbb{Z} \to \mathit{Set}\ \mathbb{Z} \to \mathit{Bool} \\
\mathit{fromList} & : [\mathbb{Z}] \to \mathit{Set}\ \mathbb{Z}
\end{array}
$$

For example, Haskell provides such a type in the library module *Data.Set*, which is implemented using size-balanced binary trees (Adams, 1993) to support the definition of the

operations in an efficient manner; e.g. the *member* function (which plays the role of *elem* for sets) takes logarithmic time. We do not require the actual definitions for our purposes here, but we will make use of three simple algebraic properties of the *Set* data type:

$$\begin{array}{ll}(a)\ \mathit{empty} & = \mathit{fromList}\ [\,] \\ (b)\ \mathit{insert}\ n\ (\mathit{fromList}\ ns) & = \mathit{fromList}\ (n :: ns) \\ (c)\ \mathit{member}\ n\ (\mathit{fromList}\ ns) & = \mathit{elem}\ n\ ns\end{array}$$

In order to improve the efficiency of *rmdups*, we now seek to transform the original function of type $[\mathbb{Z}] \to [\mathbb{Z}]$ into a worker function of type $[\mathbb{Z}] \to (\mathit{Set}\ \mathbb{Z}, [\mathbb{Z}])$ that returns the original result represented as both a set and as a list, thereby opening up the possibility of using the more efficient *member* operation on sets to decide whether or not to add a value to the result list. The necessary conversion functions between the original return type $[\mathbb{Z}]$ and the new return type $(\mathit{Set}\ \mathbb{Z}, [\mathbb{Z}])$ are defined as follows:

$$\begin{array}{lcl}\mathit{rep} & : & [\mathbb{Z}] \to (\mathit{Set}\ \mathbb{Z}, [\mathbb{Z}]) \\ \mathit{rep}\ ns & = & (\mathit{fromList}\ ns, ns) \\ \mathit{abs} & : & (\mathit{Set}\ \mathbb{Z}, [\mathbb{Z}]) \to [\mathbb{Z}] \\ \mathit{abs}\ (s, ns) & = & ns\end{array}$$

These definitions satisfy worker/wrapper assumption (A), i.e. $\mathit{abs} \circ \mathit{rep} = \mathit{id}_{[\mathbb{Z}]}$:

$$\begin{array}{ll} & \mathit{abs}\ (\mathit{rep}\ ns) \\ = & \quad \{\ \text{applying } \mathit{rep}\ \} \\ & \mathit{abs}\ (\mathit{fromList}\ ns, ns) \\ = & \quad \{\ \text{applying } \mathit{abs}\ \} \\ & ns\end{array}$$

For this example, the simplest condition to use as the basis for constructing the worker function $\mathit{work} = \mathit{fold}\ g\ v$ is condition (2), which expands to two equations:

$$\begin{array}{lcl}\mathit{rep}\ [\,] & = & v \\ \mathit{rep}\ (\mathit{add}\ n\ ns) & = & g\ n\ (\mathit{rep}\ ns)\end{array}$$

We calculate a value v satisfying the first equation as follows:

$$\begin{array}{ll} & v \\ = & \quad \{\ \text{first equation}\ \} \\ & \mathit{rep}\ [\,] \\ = & \quad \{\ \text{applying } \mathit{rep}\ \} \\ & (\mathit{fromList}\ [\,], [\,]) \\ = & \quad \{\ \text{(a)}\ \} \\ & (\mathit{empty}, [\,])\end{array}$$

In turn, we calculate a function g satisfying the second equation:

$$\begin{array}{ll} & \mathit{rep}\ (\mathit{add}\ n\ ns) \\ = & \quad \{\ \text{applying } \mathit{add}\ \} \\ & \mathit{rep}\ (\textbf{if}\ \mathit{elem}\ n\ ns\ \textbf{then}\ ns\ \textbf{else}\ n :: ns) \\ = & \quad \{\ \text{distributivity}\ \}\end{array}$$

$$
\begin{array}{ll}
 & \textbf{if } elem\ n\ ns \textbf{ then } rep\ ns \textbf{ else } rep\ (n :: ns) \\
= & \quad \{ \text{ applying } rep \ \} \\
 & \textbf{if } elem\ n\ ns \textbf{ then } (fromList\ ns, ns) \textbf{ else } (fromList\ (n :: ns), n :: ns) \\
= & \quad \{ \text{ (b) and (c) } \} \\
 & \textbf{if } member\ n\ (fromList\ ns) \textbf{ then } (fromList\ ns, ns) \textbf{ else } (insert\ n\ (fromList\ ns), n :: ns) \\
= & \quad \{ \text{ define } g\ n\ (s, ns) = \textbf{if } member\ n\ s \textbf{ then } (s, ns) \textbf{ else } (insert\ n\ s, n :: ns) \ \} \\
 & g\ n\ (fromList\ ns, ns) \\
= & \quad \{ \text{ unapplying } rep \ \} \\
 & g\ n\ (rep\ ns)
\end{array}
$$

In summary, we have calculated the following new definitions:

$$
\begin{array}{l}
work\ :\ [\mathbb{Z}] \to (Set\ \mathbb{Z}, [\mathbb{Z}]) \\
work = fold\ g\ v \\
\qquad \textbf{where} \\
\qquad\quad \begin{array}{lll}
g & : & \mathbb{Z} \to (Set\ \mathbb{Z}, [\mathbb{Z}]) \to (Set\ \mathbb{Z}, [\mathbb{Z}]) \\
g\ n\ (s, ns) & = & \textbf{if } member\ n\ s \textbf{ then } (s, ns) \textbf{ else } (insert\ n\ s, n :: ns) \\
v & : & (Set\ \mathbb{Z}, [\mathbb{Z}]) \\
v & = & (empty, [\,])
\end{array}
\end{array}
$$

Finally, applying worker/wrapper factorisation gives $rmdups = abs \circ work$, which by making the list argument explicit and expanding *abs* results in a more efficient version of *rmdups* that takes quasilinear time:

$$
\begin{array}{l}
rmdups\ :\ [\mathbb{Z}] \to [\mathbb{Z}] \\
rmdups\ ns = snd\ (work\ ns)
\end{array}
$$

We conclude with a number of remarks about this example. First of all, once we made the decision to represent the result of *rmdups* as both a set and a list, the rest of the derivation proceeds using straightforward equational reasoning, without the need to use induction or the definitions for the underlying primitive operations on sets. And secondly, we note that condition (2) encapsulates precisely the properties that are required to admit such a simple, non-inductive derivation. In particular, conditions (1) and (3) are not valid for this example, while (1β) and (2β) require the use of induction.

5.6 Example: An Evaluation Function

For our second example, we consider a type *Expr* of simple arithmetic expressions comprising integers and division, together with its associated *fold* operator:

$$
\begin{array}{ll}
\textbf{data } Expr & = Val\ \mathbb{Z} \mid Div\ Expr\ Expr \\
fold & : (\mathbb{Z} \to A) \to (A \to A \to A) \to Expr \to A \\
fold\ v\ d\ (Val\ n) & = v\ n \\
fold\ v\ d\ (Div\ e_1\ e_2) & = d\ (fold\ v\ d\ e_1)\ (fold\ v\ d\ e_2)
\end{array}
$$

Now suppose that we wish to define an evaluator for such expressions that takes explicit account of the possibility of failure due to division by zero. To this end, we first recall

the type *Maybe A* of values of type *A* that may fail, in which the value *Nothing* represents failure, and values of the form *Just x* represent success:

$$\textbf{data}\ \mathit{Maybe}\ A\ =\ \mathit{Nothing}\ \mid\ \mathit{Just}\ A$$

Using this type, we can define a division function *safediv* that checks for division by zero, and a function *maybediv* that divides two integers that may fail:

$$\begin{array}{lcl}
\mathit{safediv} & : & \mathbb{Z} \to \mathbb{Z} \to \mathit{Maybe}\ \mathbb{Z} \\
\mathit{safediv}\ n\ m & = & \textbf{if}\ m == 0\ \textbf{then}\ \mathit{Nothing}\ \textbf{else}\ \mathit{Just}\ (n \div m) \\
\mathit{maybediv} & : & \mathit{Maybe}\ \mathbb{Z} \to \mathit{Maybe}\ \mathbb{Z} \to \mathit{Maybe}\ \mathbb{Z} \\
\mathit{maybediv}\ x\ y & = & \textbf{case}\ x\ \textbf{of} \\
 & & \quad \mathit{Nothing} \to \mathit{Nothing} \\
 & & \quad \mathit{Just}\ n \to \textbf{case}\ y\ \textbf{of} \\
 & & \qquad \mathit{Nothing} \to \mathit{Nothing} \\
 & & \qquad \mathit{Just}\ m \to \mathit{safediv}\ n\ m
\end{array}$$

It is then straightforward to define an evaluator using *fold*:

$$\begin{array}{lcl}
\mathit{eval} & : & \mathit{Expr} \to \mathit{Maybe}\ \mathbb{Z} \\
\mathit{eval} & = & \mathit{fold}\ \mathit{Just}\ \mathit{maybediv}
\end{array}$$

However, because of the repeated pattern matching on values of type *Maybe* $\mathbb{Z}$ resulting from the use of *maybediv*, this definition for *eval* is inefficient. We now show how it can be transformed into a more efficient version that uses continuation-passing style (CPS) (Reynolds, 1972) to eliminate all such pattern matching. In particular, we seek to transform the original evaluation function of type $\mathit{Expr} \to \mathit{Maybe}\ \mathbb{Z}$ into a worker function of type $\mathit{Expr} \to (\mathbb{Z} \to \mathit{Maybe}\ \mathbb{Z}) \to \mathit{Maybe}\ \mathbb{Z}$, whose second argument is a continuation that will be applied to the result value of a successful evaluation. The necessary conversion functions between the original return type *Maybe* $\mathbb{Z}$, and the new return type, abbreviated by *Cont* $\mathbb{Z}$ by means of a type declaration, are defined as follows:

$$\begin{array}{lcl}
\textbf{type}\ \mathit{Cont}\ A & = & (A \to \mathit{Maybe}\ A) \to \mathit{Maybe}\ A \\
\mathit{rep} & : & \mathit{Maybe}\ \mathbb{Z} \to \mathit{Cont}\ \mathbb{Z} \\
\mathit{rep}\ \mathit{Nothing} & = & \lambda\ c \to \mathit{Nothing} \\
\mathit{rep}\ (\mathit{Just}\ n) & = & \lambda\ c \to c\ n \\
\mathit{abs} & : & \mathit{Cont}\ \mathbb{Z} \to \mathit{Maybe}\ \mathbb{Z} \\
\mathit{abs}\ f & = & f\ \mathit{Just}
\end{array}$$

It is easy to show that assumption (A), i.e. $\mathit{abs} \circ \mathit{rep} = \mathit{id}_{\mathit{Maybe}\,\mathbb{Z}}$, holds. As with the previous example, the best condition to use as the basis for constructing the worker function $\mathit{work} = \mathit{fold}\ v\ d$ is condition (2), which expands to two equations:

$$\begin{array}{lcl}
\mathit{rep}\ (\mathit{Just}\ n) & = & c\ n\ c \\
\mathit{rep}\ (\mathit{maybediv}\ x\ y)\ c & = & d\ (\mathit{rep}\ x)\ (\mathit{rep}\ y)\ c
\end{array}$$

We calculate a function v satisfying the first equation as follows:

$$\begin{array}{ll}
 & \mathit{rep}\ (\mathit{Just}\ n)\ c \\
= & \qquad \{\ \text{applying } \mathit{rep}\ \}
\end{array}$$

```
    c n
=       { define v n c = c n }
    v n c
```

In turn, we calculate a function d satisfying the second equation:

```
    rep (maybediv x y) c
=       { applying maybediv and safediv }
    rep (case x of
             Nothing → Nothing
             Just n  → case y of
                 Nothing → Nothing
                 Just m  → if m == 0 then Nothing else Just (n ÷ m)) c
=       { distribution over case and if }
    case x of
        Nothing → rep Nothing c
        Just n  → case y of
            Nothing → rep Nothing c
            Just m  → if m == 0 then rep Nothing c else rep (Just (n ÷ m)) c
=       { applying rep }
    case x of
        Nothing → Nothing
        Just n  → case y of
            Nothing → Nothing
            Just m  → if m == 0 then Nothing else c (n ÷ m)
=       { define safediv' n m c = if m == 0 then Nothing else c (n ÷ m) }
    case x of
        Nothing → Nothing
        Just n  → case y of
            Nothing → Nothing
            Just m  → safediv' n m c
=       { unapplying rep }
    case x of
        Nothing → Nothing
        Just n  → rep y (λ m → safediv' n m c)
=       { unapplying rep }
    rep x (λ n → rep y (λ m → safediv' n m c))
=       { define d f g c = f (λ n → g (λ m → safediv' n m c)) }
    d (rep x) (rep y) c
```

In summary, we have calculated the following new definitions, in which the continuation arguments c have been moved into the bodies of the definitions as λ-bound variables in order to make the types easier to comprehend:

```
safediv' : ℤ → ℤ → Cont ℤ
safediv' n m = λ c → if m == 0 then Nothing else c (n ÷ m)
```

$$
\begin{array}{l}
work \;:\; Expr \to Cont\,\mathbb{Z} \\
work \;=\; fold\; v\; d \\
\qquad \textbf{where}\; v \quad\;\; :\; \mathbb{Z} \to Cont\,\mathbb{Z} \\
\qquad\qquad\quad v\,n \;\; = \lambda\, c \to c\,n \\
\qquad\qquad\quad d \quad\;\; :\; Cont\,\mathbb{Z} \to Cont\,\mathbb{Z} \to Cont\,\mathbb{Z} \\
\qquad\qquad\quad d\,f\,g \;= \lambda\, c \to f\,(\lambda\, n \to g\,(\lambda\, m \to safediv'\; n\; m\; c))
\end{array}
$$

Finally, applying worker/wrapper factorisation gives $eval = abs \circ work$, which by making the expression argument explicit and expanding *abs* results in a more efficient evaluation function that does not perform any pattern matching on values of type $Maybe\,\mathbb{Z}$:

$$
\begin{array}{l}
eval \;\; :\; Expr \to Maybe\,\mathbb{Z} \\
eval\; e \;=\; work\; e\; Just
\end{array}
$$

Once again, note that the derivation proceeds using simple equational reasoning and does not require the use of induction. Moreover, in contrast to the previous derivation of an evaluation function that may fail using the worker/wrapper theory for *fix* and condition (1β) (Gill & Hutton, 2009), using the worker/wrapper theory for *fold* and condition (2) as the starting point results in a simpler derivation.

6 From Least Fixed Points to Initial Algebras

In Section 5 we developed the worker/wrapper theory for initial algebras. Given that the results were formulated for an arbitrary category $\mathbb{C}$, we would expect them to hold in the category **CPO** of cpos and continuous functions used in the least-fixed-point approach. This is indeed the case, with one complicating factor: when **CPO** is the base category, the universal property has a strictness side condition, which weakens our results by adding many strictness requirements. In this section, we show that all but one of these strictness conditions is unnecessary, by instantiating our theory for least fixed points.

6.1 Strictness

Recall that the basic proof technique for the fold operator is its universal property. In the category **CPO**, this property has a strictness side condition (Meijer *et al.*, 1991)[4]:

Lemma 6.1 (*Universal Property of Fold in* **CPO**)
If h is strict, then:

$$h = fold\, f \;\Leftrightarrow\; h \circ in = f \circ F\, h$$

The universal property of fold, together with derived properties such as fusion and the rolling rule, form the basis of our proofs of worker/wrapper factorisation and fusion for initial algebras in Section 5. Tracking the impact of the extra strictness condition above on these results is straightforward but tedious, so we omit the details here (they are provided

[4] There are other variants of the universal property in **CPO**, but they all require extra conditions concerning the undefined value $\perp$ (Fokkinga & Meijer, 1991).

in the supplementary Agda proofs) and just present the results: for conditions (1), (1β), (2) and (2β), both factorisation and fusion require that f, *rep* and *abs* are strict, while for (3), factorisation requires that g and *abs* are strict.

In summary, instantiating the worker/wrapper results for initial algebras to the category **CPO** is straightforward, but deriving the results in this manner introduces many strictness side conditions that may limit their applicability. Some of these conditions could be avoided by using more liberal versions of derived properties such as fold fusion and the rolling rule that are proved from first principles rather than being derived from the universal property. However, it turns out that most of the strictness conditions can be avoided using our worker/wrapper theory for least fixed points.

6.2 From Fix to Fold

As noted earlier, the generalised worker/wrapper results for initial algebras are very similar to those for least fixed points. Indeed, unifying the results in this manner is one of the primary contributions of this article. In this section we show how the initial-algebra results in **CPO** can in fact be derived from those for least fixed points, by exploiting the fact that in this context *fold* can be defined in terms of *fix* (Meijer *et al.*, 1991):

Lemma 6.2 (*Definition of Fold using Fix in* **CPO**)

$$fold\, f \;=\; fix\, (\lambda\, h \;\to\; f \circ F\, h \circ out)$$

Suppose we are given algebras $f : F\, A \to A$ and $g : F\, B \to B$, and conversion functions $rep : A \to B$ and $abs : B \to A$. Our aim is to use the worker/wrapper results for *fix* to derive assumptions and conditions that imply the factorisation result for *fold*, that is:

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

First, we define functions f' and g' such that $fold\, f = fix\, f'$ and $fold\, g = fix\, g'$:

$$\begin{array}{ll} f' \;:\; (\mu F \to A) \to (\mu F \to A) & g' \;:\; (\mu F \to B) \to (\mu F \to B) \\ f' \;=\; \lambda\, h \to f \circ F\, h \circ out & g' \;=\; \lambda\, h \to g \circ F\, h \circ out \end{array}$$

Then we define conversion functions between the types for $fold\, f$ and $fold\, g$:

$$\begin{array}{ll} rep' \;:\; (\mu F \to A) \to (\mu F \to B) & abs' \;:\; (\mu F \to B) \to (\mu F \to A) \\ rep'\, h \;=\; rep \circ h & abs'\, h \;=\; abs \circ h \end{array}$$

Using these definitions, the worker/wrapper equation $fold\, f = abs \circ fold\, g$ in terms of *fold* is equivalent to the following equation in terms of *fix*:

$$fix\, f' \;=\; abs'\, (fix\, g')$$

This equation has the form of worker/wrapper factorisation for fix, and is hence valid provided that we have one of the assumptions

(A) $abs' \circ rep' \;=\; id_{\mu F \to A}$

(B) $abs' \circ rep' \circ f' \;=\; f'$

(C) $fix\, (abs' \circ rep' \circ f') \;=\; fix\, f'$

together with one of the conditions

(1) $g' = rep' \circ f' \circ abs'$ (1β) $fix\, g' = fix\, (rep' \circ f' \circ abs')$

(2) $rep' \circ f' = g' \circ rep' \wedge \text{strict } rep'$ (2β) $fix\, g' = rep'\, (fix\, f')$

(3) $abs' \circ g' = f' \circ abs'$

By expanding definitions, it is now straightforward to simplify each of these assumptions and conditions in terms of the original functions f, g, *rep* and *abs*.

Assumption (A):

$$
\begin{array}{ll}
 & abs' \circ rep' = id_{\mu F \to A} \\
\Leftrightarrow & \quad \{ \text{ equality for functions, composition } \} \\
 & \forall\, h.\, abs'\, (rep'\, h) = h \\
\Leftrightarrow & \quad \{ \text{ applying } abs' \text{ and } rep' \} \\
 & \forall\, h.\, abs \circ rep \circ h = h \\
\Leftarrow & \quad \{ \text{ extensionality } \} \\
 & abs \circ rep = id_A
\end{array}
$$

Assumption (B):

$$
\begin{array}{ll}
 & abs' \circ rep' \circ f' = f' \\
\Leftrightarrow & \quad \{ \text{ equality for functions, composition } \} \\
 & \forall\, h.\, abs'\, (rep'\, (f'\, h)) = f'\, h \\
\Leftrightarrow & \quad \{ \text{ applying } abs', rep' \text{ and } f' \} \\
 & \forall\, h.\, abs \circ rep \circ f \circ F\, h \circ out = f \circ F\, h \circ out \\
\Leftarrow & \quad \{ \text{ extensionality } \} \\
 & abs \circ rep \circ f = f
\end{array}
$$

Assumption (C):

$$
\begin{array}{ll}
 & fix\, (abs' \circ rep' \circ f') = fix\, f' \\
\Leftrightarrow & \quad \{ \text{ applying } abs', rep' \text{ and } f', \text{ simplification } \} \\
 & fix\, (\lambda\, h \to abs \circ rep \circ f \circ F\, h \circ out) = fix\, (\lambda\, h \to f \circ F\, h \circ out) \\
\Leftrightarrow & \quad \{ \text{ unapplying } fold \} \\
 & fold\, (abs \circ rep \circ f) = fold\, f
\end{array}
$$

Condition (1):

$$
\begin{array}{ll}
 & g' = rep' \circ f' \circ abs' \\
\Leftrightarrow & \quad \{ \text{ equality for functions, composition } \} \\
 & \forall\, h.\, g'\, h = rep'\, (f'\, (abs'\, h)) \\
\Leftrightarrow & \quad \{ \text{ applying } g', rep', f' \text{ and } abs' \} \\
 & \forall\, h.\, g \circ F\, h \circ out = rep \circ f \circ F\, (abs \circ h) \circ out \\
\Leftrightarrow & \quad \{ \text{ functors } \} \\
 & \forall\, h.\, g \circ F\, h \circ out = rep \circ f \circ F\, abs \circ F\, h \circ out \\
\Leftarrow & \quad \{ \text{ extensionality } \} \\
 & g = rep \circ f \circ F\, abs
\end{array}
$$

Condition (1β):

$$
\begin{array}{ll}
 & \mathit{fix}\ g' = \mathit{fix}\ (\mathit{rep}' \circ f' \circ \mathit{abs}') \\
\Leftrightarrow & \quad \{ \text{ applying } g', \mathit{rep}', f' \text{ and } \mathit{abs}', \text{ simplification } \} \\
 & \mathit{fix}\ (\lambda\, h \to g \circ F\, h \circ \mathit{out}) = \mathit{fix}\ (\lambda\, h \to \mathit{rep} \circ f \circ F\ (\mathit{abs} \circ h) \circ \mathit{out}) \\
\Leftrightarrow & \quad \{ \text{ functors } \} \\
 & \mathit{fix}\ (\lambda\, h \to g \circ F\, h \circ \mathit{out}) = \mathit{fix}\ (\lambda\, h \to \mathit{rep} \circ f \circ F\, \mathit{abs} \circ F\, h \circ \mathit{out}) \\
\Leftrightarrow & \quad \{ \text{ unapplying } \mathit{fold} \} \\
 & \mathit{fold}\ g = \mathit{fold}\ (\mathit{rep} \circ f \circ F\, \mathit{abs})
\end{array}
$$

Condition (2):

$$
\begin{array}{ll}
 & \mathit{rep}' \circ f' = g' \circ \mathit{rep}' \\
\Leftrightarrow & \quad \{ \text{ equality for functions, composition } \} \\
 & \forall\, h.\ \mathit{rep}'\ (f'\ h) = g'\ (\mathit{rep}'\ h) \\
\Leftrightarrow & \quad \{ \text{ applying } \mathit{rep}', f' \text{ and } g' \} \\
 & \forall\, h.\ \mathit{rep} \circ f \circ F\, h \circ \mathit{out} = g \circ F\ (\mathit{rep} \circ h) \circ \mathit{out} \\
\Leftrightarrow & \quad \{ \text{ functors } \} \\
 & \forall\, h.\ \mathit{rep} \circ f \circ F\, h \circ \mathit{out} = g \circ F\, \mathit{rep} \circ F\, h \circ \mathit{out} \\
\Leftarrow & \quad \{ \text{ extensionality } \} \\
 & \mathit{rep} \circ f = g \circ F\, \mathit{rep}
\end{array}
$$

and

$$
\begin{array}{ll}
 & \text{strict } \mathit{rep}' \\
\Leftrightarrow & \quad \{ \text{ applying } \mathit{rep}' \} \\
 & \mathit{strict}\ (\lambda\, h \to \mathit{rep} \circ h) \\
\Leftrightarrow & \quad \{ \text{ strictness } \} \\
 & (\lambda\, h \to \mathit{rep} \circ h)\ \bot = \bot \\
\Leftrightarrow & \quad \{ \beta\text{-reduction } \} \\
 & \mathit{rep} \circ \bot = \bot \\
\Leftrightarrow & \quad \{ \text{ strictness } \} \\
 & \text{strict } \mathit{rep}
\end{array}
$$

Condition (2β):

$$
\begin{array}{ll}
 & \mathit{fix}\ g' = \mathit{rep}'\ (\mathit{fix}\, f') \\
\Leftrightarrow & \quad \{ \text{ applying } g', \mathit{rep}' \text{ and } f' \} \\
 & \mathit{fix}\ (\lambda\, h \to g \circ F\, h \circ \mathit{out}) = \mathit{rep} \circ \mathit{fix}\ (\lambda\, h \to f \circ F\, h \circ \mathit{out}) \\
\Leftrightarrow & \quad \{ \text{ unapplying } \mathit{fold} \} \\
 & \mathit{fold}\ g = \mathit{rep} \circ \mathit{fold}\, f
\end{array}
$$

Condition (3):

$$
\begin{array}{ll}
 & \mathit{abs}' \circ g' = f' \circ \mathit{abs}' \\
\Leftrightarrow & \quad \{ \text{ equality for functions, composition } \} \\
 & \forall\, h.\ \mathit{abs}'\ (g'\ h) = f'\ (\mathit{abs}'\ h)
\end{array}
$$

Given functions

$$f : F\,A \to A$$
$$g : F\,B \to B$$

for some functor F, and conversion functions

$$rep : A \to B$$
$$abs : B \to A$$

then we have a set of *worker/wrapper assumptions*

(A) $abs \circ rep = id_A$
(B) $abs \circ rep \circ f = f$
(C) $fold\,(abs \circ rep \circ f) = fold\,f$

and a set of *worker/wrapper conditions*

(1) $g = rep \circ f \circ F\,abs$ (1β) $fold\,g = fold\,(rep \circ f \circ F\,abs)$
(2) $rep \circ f = g \circ F\,rep \wedge$ strict rep (2β) $fold\,g = rep \circ fold\,f$
(3) $abs \circ g = f \circ F\,abs$

Provided that any of the assumptions hold and any of the conditions hold, then *worker/wrapper* factorisation is valid:

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

Furthermore, if any of the assumptions hold, and any of the conditions except (3) hold, then *worker/wrapper fusion* is valid:

$$rep \circ abs \circ fold\,g = fold\,g$$

Figure 3: Worker/wrapper transformation for initial algebras in **CPO**.

$$\begin{array}{ll}
\Leftrightarrow & \{\ \text{applying } abs', g', \text{and } f'\ \} \\
 & \forall\, h.\ abs \circ g \circ F\,h \circ out = f \circ F\,(abs \circ h) \circ out \\
\Leftrightarrow & \{\ \text{functors}\ \} \\
 & \forall\, h.\ abs \circ g \circ F\,h \circ out = f \circ F\,abs \circ F\,h \circ out \\
\Leftarrow & \{\ \text{extensionality}\ \} \\
 & abs \circ g = f \circ F\,abs
\end{array}$$

Some of the above calculations that are presented as implications can also be strengthened to equivalences, but for the purposes of simplifying the assumptions and conditions we only require implications so have not done so here. The resulting worker/wrapper-factorisation theorem for initial algebras in the category **CPO** is summarised in Figure 3. We can also apply the same procedure to worker/wrapper fusion, as by applying definitions the fusion equation in terms of *fold* expands to the form of the fusion equation in terms of *fix*:

$$\begin{array}{ll}
 & rep \circ abs \circ fold\,g = fold\,g \\
\Leftrightarrow & \{\ \text{applying } rep, abs \text{ and } g\ \} \\
 & rep'\ (abs'\ (fix\ g')) = fix\ g'
\end{array}$$

Hence, provided that we are given any of the assumptions (A), (B) or (C) and any of the conditions except (3) from Figure 3, then:

$$\mathit{rep} \circ \mathit{abs} \circ \mathit{fold}\ g\ =\ \mathit{fold}\ g$$

In conclusion, we now have an alternative derivation of the worker/wrapper results for initial algebras in **CPO**. Compared to the derivation in Section 6.1, this new approach eliminated all the strictness conditions except for *rep* being strict in condition (2), and is hence more generally applicable. One might ask if we can also drop strictness from (2), but the answer is no. In order to verify this, let us take $\mathit{Id} : \mathbf{CPO} \to \mathbf{CPO}$ as the identity functor, for which it can be shown by fixed-point induction that $\mathit{fold}\ f\ \bot = \mathit{fix}\ f$. Now consider the example from Section 3.2 that shows that strictness cannot be dropped from (2) in the theory for *fix*. This example satisfies (A) and $\mathit{rep} \circ f = g \circ \mathit{Id}\ \mathit{rep}$, but not worker/wrapper factorisation $\mathit{fold}\ f\ =\ \mathit{abs} \circ \mathit{fold}\ g$. In particular, if we assume factorisation is valid we could apply both sides to $\bot$ to obtain $\mathit{fold}\ f\ \bot\ =\ \mathit{abs}\ (\mathit{fold}\ g\ \bot)$, which by the above result is equivalent to $\mathit{fix}\ f = \mathit{abs}\ (\mathit{fix}\ g)$, which does not hold for this example as shown in Section 3.2. Hence, by contradiction, $\mathit{fold}\ f = \mathit{abs} \circ \mathit{fold}\ g$ is invalid.

7 Related Work

A historical review of the worker/wrapper transformation and related work was given in Gill & Hutton (2009), so we direct the reader to that article rather than repeating the details here. The transformation can also be viewed as a form of *data refinement* (Hoare, 1972; Morgan & Gardiner, 1990), a general-purpose approach to replacing a data structure by a more efficient version. Specifically, the worker/wrapper transformation is a data refinement technique for functional programs defined using the recursion operators *fix* or *fold*.

Recently, Gammie (2011) observed that the manner in which the worker/wrapper-fusion rule was used in Gill & Hutton (2009) may lead to the introduction of non-termination. However, this is a well-known consequence of the fold/unfold approach to program transformation (Burstall & Darlington, 1977; Tullsen, 2002), which in general only preserves partial correctness, rather than being a problem with the fusion rule itself, which is correct as it stands. Alternative, but less expressive, transformation frameworks that guarantee total correctness have been proposed, such as the use of expression procedures (Scherlis, 1980). Gammie's solution was to add the requirement that *rep* be strict to the worker/wrapper-fusion rule, which holds for the relevant examples in the original article. However, we have not added this requirement in the present article, as this would unnecessarily weaken the fusion rule without overcoming the underlying issue with fold/unfold transformation. Gammie also pointed out that the stream memoisation example in Gill & Hutton (2009) incorrectly claims that assumption (A) holds, but we note that the example as a whole is still correct as the weaker assumption (B) does hold.

In this article we have focused on developing the theory of the worker/wrapper transformation, with the aim of making it as widely applicable as possible. Meanwhile, a team in Kansas is putting the technique into mechanised practice as part of the HERMIT project (Farmer *et al.*, 2012). In particular, they are developing a general purpose system for optimising Haskell programs that allows programmers to write sufficient annotations to permit the Glasgow Haskell Compiler to apply custom transformations automatically. The

worker/wrapper transformation was the first high-level technique encoded in the system, and it then proved relatively straightforward to mechanise a selection of new and existing worker/wrapper examples (Sculthorpe *et al.*, 2013). Working with the automated system has also revealed that other, more specialised, transformation techniques can be cast as instances of worker/wrapper, and consequently that using the worker/wrapper infrastructure can simplify mechanising those transformations (Sculthorpe *et al.*, 2013).

8 Conclusions and Future Work

The original worker/wrapper article (Gill & Hutton, 2009) formalised the basic technique using least fixed points, while the follow-up article (Hutton *et al.*, 2010) developed a worker/wrapper theory for initial algebras. In this article we showed how the two approaches can be generalised in a uniform manner by combining their different sets of correctness conditions. Moreover, we showed how the new theories can be further generalised with conditions that are both necessary and sufficient to ensure the correctness of the transformations. All the proofs have been mechanically checked using the Agda proof assistant, and are available as supplementary material on the JFP website.

It is interesting to recount how the conditions (1β) and (2β) were developed. Initially we focused on combining assumptions (A), (B) and (C) from the first article with conditions (1), (2) and (3) from the second. However, the resulting theory was still not powerful enough to handle some examples that we intuitively felt should fit within the framework. It was only when we looked again at the proofs for worker/wrapper factorisation and fusion that we realised that conditions (1) and (2) could be further weakened, resulting in conditions (1β) and (2β), and proofs that they are equivalent and maximally general.

In terms of further work, practical applications of the worker/wrapper technique are being driven forward by the HERMIT project in Kansas, as described in Section 7. On the foundational side, it would be interesting to exploit additional forms of structure to further extend the generality and applicability of the technique, for example by using other recursion operators such as unfolds and hylomorphisms, framing the technique using more general categorical constructions such as limits and colimits, and considering more sophisticated notions of computation such as monadic, comonadic and applicative programs.

Acknowledgements

The first author was supported by NSF award number 1117569. We would like to thank Nicolas Frisby and Andy Gill for useful discussions on the Fibonacci example, Andy Gill for the *rmdups* example, Jennifer Hackett for the counterexample in Section 6, and the anonymous referees for their detailed and helpful reviews. The title of this article is inspired by the lyrics from Bangalter *et al.* (2001).

References

Adams, Stephen. (1993). Efficient sets — a balancing act. *Journal of Functional Programming*, **3**(4), 553–561.

Backhouse, Roland. (2002). Galois Connections and Fixed Point Calculus. *Pages 89–150 of: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. Springer.

Bangalter, Thomas, de Homem-Christo, Guy-Manuel, & Birdsong, Edwin. (2001). Harder, Better, Faster, Stronger. *Daft Punk: Discovery*. United Kingdom: Virgin Records.

Bird, Richard, & de Moor, Oege. (1997). *Algebra of Programming*. Prentice Hall.

Burstall, Rod. M., & Darlington, John. (1977). A Transformation System for Developing Recursive Programs. *Journal of the ACM*, **24**(1), 44–67.

Farmer, Andrew, Gill, Andy, Komp, Ed, & Sculthorpe, Neil. (2012). The HERMIT in the Machine: A Plugin for the Interactive Transformation of GHC Core Language Programs. *Pages 1–12 of: Haskell Symposium*. ACM.

Fokkinga, Maarten M., & Meijer, Erik. (1991). *Program Calculation Properties of Continuous Algebras*. Tech. rept. CS-R9104. CWI, Amsterdam, Netherlands.

Gammie, Peter. (2011). Strict Unwraps Make Worker/Wrapper Fusion Totally Correct. *Journal of Functional Programming*, **21**(2), 209–213.

Gill, Andy, & Hutton, Graham. (2009). The Worker/Wrapper Transformation. *Journal of Functional Programming*, **19**(2), 227–251.

Hoare, Tony. (1972). Proof of Correctness of Data Representations. *Acta Informatica*, **1**(4), 271–281.

Hutton, Graham, Jaskelioff, Mauro, & Gill, Andy. (2010). Factorising Folds for Faster Functions. *Journal of Functional Programming*, **20**(3&4), 353–373.

Meijer, Erik, Fokkinga, Maarten M., & Paterson, Ross. (1991). Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. *Pages 124–144 of: Functional Programming Languages and Computer Architecture*. Springer.

Morgan, Carroll, & Gardiner, P. H. B. (1990). Data Refinement by Calculation. *Acta Informatica*, **27**(6), 481–503.

Peyton Jones, Simon. (2007). Call-pattern Specialisation for Haskell Programs. *Pages 327–337 of: International Conference on Functional Programming*. ACM.

Peyton Jones, Simon, & Launchbury, John. (1991). Unboxed Values as First Class Citizens in a Non-Strict Functional Language. *Pages 636–666 of: Functional Programming Languages and Computer Architecture*. Springer.

Reynolds, John C. (1972). Definitional Interpreters for Higher-Order Programming Languages. *Pages 717–740 of: ACM Annual Conference*. ACM.

Scherlis, William Louis. (1980). *Expression Procedures and Program Derivation*. Ph.D. thesis, Stanford University.

Schmidt, David A. (1986). *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon.

Sculthorpe, Neil, Farmer, Andrew, & Gill, Andy. (2013). The HERMIT in the Tree: Mechanizing Program Transformations in the GHC Core Language. *Pages 86–103 of: Implementation and Application of Functional Languages 2012*. Lecture Notes in Computer Science, vol. 8241. Springer.

Tullsen, Mark. (2002). *PATH, A Program Transformation System for Haskell*. Ph.D. thesis, Yale University.

Voigtländer, Janis. (2008). Asymptotic Improvement of Computations over Free Monads. *Pages 388–403 of: Mathematics of Program Construction*. Lecture Notes in Computer Science, vol. 5133. Springer.

Wadler, Philip. (1989). Theorems for Free! *Pages 347–359 of: Functional Programming and Computer Architecture*. Springer.

Winskel, Glynn. (1993). *The Formal Semantics of Programming Languages — An Introduction*. Foundation of Computing. MIT.