JFP **24** (1): 113–127, 2014. © Cambridge University Press 2014
doi:10.1017/S0956796814000045

Work it, wrap it, fix it, fold it

NEIL SCULTHORPE
University of Kansas, USA
(*e-mail:* neil@ittc.ku.edu)

GRAHAM HUTTON
University of Nottingham, UK
(*e-mail:* Graham.Hutton@nottingham.ac.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 paper 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 paper 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 20 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 (Gill & Hutton 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. 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 (Hutton *et al.* 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 paper is to show how this can be achieved, and to explore the benefits that result. More precisely, the paper 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 paper is aimed at readers who are familiar with the basics of a least-fixed-point semantics (Schmidt 1986), initial-algebra semantics (Bird & de Moor 1997), and the worker/wrapper transformation (Gill & Hutton 2009; Hutton *et al.* 2010), but all necessary concepts and results are reviewed. A mechanical verification of the proofs in Agda is available as online supplementary material at dx.doi.org/10.1017/S0956796814000045, along with an extended version of this paper that includes a series of worked examples and all proofs.

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

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

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).

A *complete partial order* (cpo) is a set with a partial-ordering $\sqsubseteq$, a least element $\perp$, and limits $\bigsqcup$ (least upper bounds) of all non-empty chains. A function f between cpos is *continuous* if it is monotonic and preserves the limit structure. If it also preserves the least element, i.e. $f \perp = \perp$, the function is *strict*. 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 $\mathit{fix}\, f$.

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

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

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 2.2 (*Fixed-Point Fusion*)

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

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 2.3 (*Rolling Rule*)

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

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 an 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).

In addition, we require one of the following *worker/wrapper conditions*[2] that relate the body functions f and g of the original and worker programs:

(1) $g = \mathit{rep} \circ f \circ \mathit{abs}$ (1β) $\mathit{fix}\,g = \mathit{fix}\,(\mathit{rep} \circ f \circ \mathit{abs})$
(2) $\mathit{rep} \circ f = g \circ \mathit{rep} \wedge \text{strict}\,\mathit{rep}$ (2β) $\mathit{fix}\,g = \mathit{rep}\,(\mathit{fix}\,f)$
(3) $\mathit{abs} \circ g = f \circ \mathit{abs}$

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) ⇒ (1β) and (2) ⇒ (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. 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 $\mathit{fix}\,f = \mathit{abs}\,(\mathit{fix}\,g)$, which there is no merit in considering as this is precisely the worker/wrapper factorisation result that we wish to establish.

In terms of how the worker/wrapper conditions are used in practice, for some applications the worker program $\mathit{fix}\,g$ 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 does not involve the use of *fix*. For other applications, our aim will be to *construct* the

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

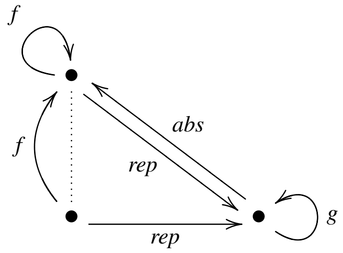
worker program. In such cases, condition (1), (1β), or (2β) provides explicit but inefficient definitions for the worker program in terms of the body function f of the original program, which we then attempt to make more efficient using program-fusion techniques. This was the approach that was taken by Gill & Hutton (2009). However, as shown by Hutton *et al.* (2010), in some cases it is preferable to use condition (2) or (3), which provides 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). For condition (1β), factorisation is verified by the following simple calculation:

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

For condition (3), at first glance it may appear that we do not need assumption (C) at all, as condition (3) on its own is sufficient to verify the result by fusion. But the use of fusion requires that *abs* is strict. However, using assumption (C) and fixed-point induction, we can prove the factorisation result without this extra strictness condition (see the extended version of this paper for details). But perhaps *abs* being strict is implied by the assumptions and conditions? In fact, given assumption (A), this is indeed the case. 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*

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

and a set of *worker/wrapper conditions*

(1) $g = rep \circ f \circ abs$ (1β) $\mathit{fix}\, g = \mathit{fix}\,(rep \circ f \circ abs)$
(2) $rep \circ f = g \circ rep \wedge$ strict rep (2β) $\mathit{fix}\, g = rep\,(\mathit{fix}\, f)$
(3) $abs \circ g = f \circ abs$

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

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

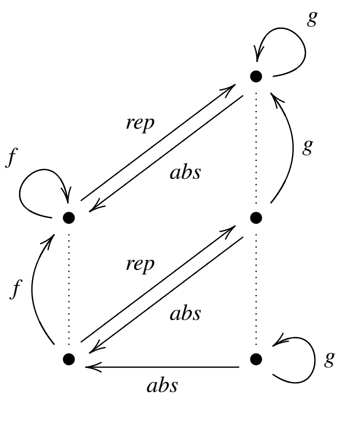
Furthermore, if any of the assumptions hold, and any of the conditions except (3) hold, then *worker/wrapper fusion* is valid:

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

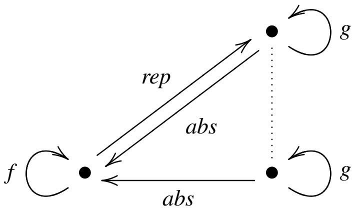
Fig. 1. Worker/wrapper transformation for least fixed points.

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) ⇒ (2β) using fixed-point fusion requires 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 $rep \circ f = g \circ rep$ 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 $rep \circ f = g \circ rep$ 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 $rep \circ f = g \circ rep$, assumption (A), 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

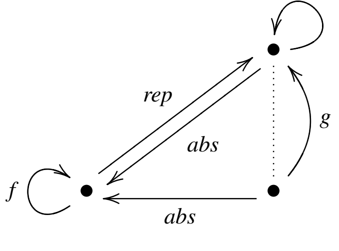
When applying worker/wrapper factorisation, it is often desirable to fuse together instances of the conversion functions *rep* and *abs* to eliminate the overhead of repeatedly converting between the new and original types (Gill & Hutton 2009). In general, it is not the case that $rep \circ abs$ can 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:

$$rep\ (abs\ (\mathit{fix}\ g)) = \mathit{fix}\ g$$

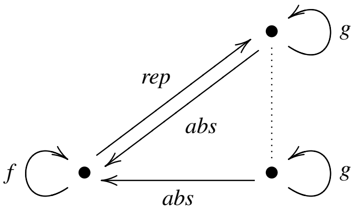
In a similar manner to Section 3.2, for the purposes of proving this result, it suffices to consider assumption (C) and condition (2β):

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

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 $rep \circ f = g \circ rep$, assumption (A), and worker/wrapper fusion:



Finally, in the case of condition (3), the following example shows that conditon (3) and assumption (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 with 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 was not 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 conditions (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:

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

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.

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).

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 $h \circ f \;=\; g \circ F\,h$. 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 $(\mu F, \mathit{in})$ for an initial F-algebra, and $\mathit{fold}\,f$ for the unique homomorphism from this initial algebra to any other algebra (A,f). Moreover, the arrow $\mathit{in} : F\;\mu F \to \mu F$ has an inverse $\mathit{out} : \mu F \to F\;\mu F$, which establishes an isomorphism $F\;\mu F \cong \mu F$. The above definition for $\mathit{fold}\,f$ can also be expressed as the following equivalence, known as the *universal property of fold*.

Lemma 4.1 (*Universal Property of Fold*)

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

The universal property forms the basic proof technique for the fold operator. For example, it can be used to verify the corresponding versions of fixed-point fusion (Lemma 2.2) and the rolling rule (Lemma 2.3) for initial algebras.

Lemma 4.2 (*Fold Fusion*)

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

Lemma 4.3 (*Rolling Rule*)

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

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. The properties that we use to validate the factorisation equation are similar to those that we identified for least fixed points, and are summarised in Figure 2. As previously, the assumptions form an hierarchy (A) ⇒ (B) ⇒ (C), the conditions (1β) and (2β) are weaker versions of (1) and (2) and are equivalent given assumption (C), and in general there is no relationship between conditions (1), (2), and (3). 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.

Worker/wrapper fusion can also be formulated for initial algebras, as shown in Figure 2. Moreover, the example from Section 3.3 that shows that fusion is not in general valid for condition (3) for least fixed points can readily be adapted to the case of initial algebras.

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

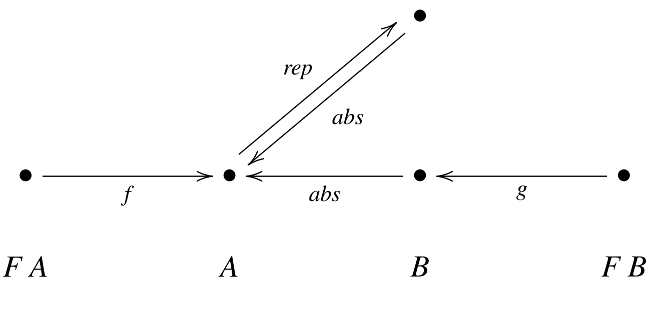
$$fold\,f = abs \circ fold\,g$$

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

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

Fig. 2. Worker/wrapper transformation for 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 condition (3) and assumption (A) but not worker/wrapper fusion:



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 condition (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 condition (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.

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).

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

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

The universal property of fold, together with derived properties such as fusion and the rolling rule, forms 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 (these are provided 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 paper. 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**)

$$\mathit{fold}\, f \;=\; \mathit{fix}\, (\lambda\, h \;\rightarrow\; f \;\circ\; F\, h \;\circ\; \mathit{out})$$

Suppose we are given algebras $f : F\, A \rightarrow A$ and $g : F\, B \rightarrow B$, and conversion functions $\mathit{rep} : A \rightarrow B$ and $\mathit{abs} : B \rightarrow A$. 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:

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

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

$$\begin{aligned} f' &: (\mu F \rightarrow A) \rightarrow (\mu F \rightarrow A) & \qquad g' &: (\mu F \rightarrow B) \rightarrow (\mu F \rightarrow B) \\ f' &= \lambda\, h \rightarrow f \circ F\, h \circ \mathit{out} & g' &= \lambda\, h \rightarrow g \circ F\, h \circ \mathit{out} \end{aligned}$$

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

$$\begin{aligned} \mathit{rep}' &: (\mu F \rightarrow A) \rightarrow (\mu F \rightarrow B) & \qquad \mathit{abs}' &: (\mu F \rightarrow B) \rightarrow (\mu F \rightarrow A) \\ \mathit{rep}'\, h &= \mathit{rep} \circ h & \mathit{abs}'\, h &= \mathit{abs} \circ h \end{aligned}$$

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

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

This equation has the form of worker/wrapper factorisation for *fix*, and is hence valid, provided one of the assumptions and one of the conditions from Figure 1 are satisfied for f', g', rep', and 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* (see the extended version of this paper for details). A similar procedure can be applied to worker/wrapper fusion. The end result is a worker/wrapper theory for initial algebras in **CPO** that has the same form as Figure 2, except that condition (2) requires that *rep* is strict. Compared with the derivation in Section 6.1, this new approach eliminates all but one strictness requirement, and hence the resulting theory is more generally applicable.

One might ask if we can also drop strictness from condition (2), but the answer is no. In order to verify this, let us take $\mathit{Id} : \mathbf{CPO} \rightarrow \mathbf{CPO}$ 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 works

An 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 (2011) 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 paper, as this would unnecessarily weaken the fusion rule without overcoming the underlying issue with fold/unfold transformation. Gammie (2011) 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 paper 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 paper 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 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.

Acknowledgments

The first author was supported by NSF award number 1117569. We would like to thank Jennifer Hackett for the counterexample in Section 6, and anonymous referees for their detailed and helpful reviews.

References

Backhouse, R. (2002) Galois connections and fixed point calculus. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*, Backhouse, R., Crole, R. & Gibbons, J. (eds). New York, NY: Springer, pp. 89–150.

Bird, R. & de Moor, O. (1997) *Algebra of Programming*. Upper Saddle River, NJ: Prentice Hall.

Burstall, R. M., & Darlington, J. (1977) A transformation system for developing recursive programs. *J. ACM* **24**(1), 44–67.

Farmer, A., Gill, A., Komp, E. & Sculthorpe, N. (2012) The HERMIT in the machine: A plugin for the interactive transformation of GHC core language programs. In *Proceedings of the 2012 Haskell Symposium*, Copenhagen, Denmark. New York, NY: ACM, pp. 1–12.

Gammie, P. (2011) Strict unwraps make worker/wrapper fusion totally correct. *J. Funct. Program.* **21**(2), 209–213.

Gill, A. & Hutton, G. (2009) The worker/wrapper transformation. *J. Funct. Program.* **19**(2), 227–251.

Hoare, T. (1972) Proof of correctness of data representations. *Acta Inform.* **1**(4), 271–281.

Hutton, G., Jaskelioff, M., & Gill, A. (2010) Factorising folds for faster functions. *J. Funct. Program.* **20**(3 & 4), 353–373.

Meijer, E., Fokkinga, M. M., & Paterson, R. (1991) Functional programming with bananas, lenses, envelopes and barbed wire. In *Functional Programming Languages and Computer Architecture*, Hughes, J. (ed). Berlin, Germany: Springer, pp. 124–144.

Morgan, C. & Gardiner, P. H. B. (1990) Data refinement by calculation. *Acta Inform.* **27**(6), 481–503.

Peyton Jones, S. & Launchbury, J. (1991) Unboxed values as first class citizens in a non-strict functional language. In *Functional Programming Languages and Computer Architecture*, Hughes, J. (ed). Berlin, Germany: Springer, pp. 636–666.

Scherlis, W. L. (1980) *Expression Procedures and Program Derivation*. PhD thesis, Stanford University, Stanford, CA.

Schmidt, D. A. (1986) *Denotational Semantics: A Methodology for Language Development*. Boston, MA: Allyn & Bacon.

Sculthorpe, N., Farmer, A. & Gill, A. (2013) The HERMIT in the tree: Mechanizing program transformations in the GHC core language. In *Implementation and Application of Functional Languages 2012*, Hinze, R. (ed), Lecture Notes in Computer Science, vol. 8241. New York, NY: Springer, pp. 86–103.

Tullsen, M. (2002) *PATH, A Program Transformation System for Haskell*. PhD thesis, Yale University, Yale. CT.

Voigtländer, J. (2008) Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction*, Audebaud, P. & Paulin-Mohring, C. (eds), Lecture Notes in Computer Science, vol. 5133. New York, NY: Springer, pp. 388–403.

Winskel, G. (1993) *The Formal Semantics of Programming Languages – An Introduction*. Foundation of Computing series. Cambridge, MA: MIT.