Programs for Cheap!

Jennifer Hackett and Graham Hutton
School of Computer Science
University of Nottingham, UK

***Abstract*—Write down the definition of a recursion operator on a piece of paper. Tell me its type, but be careful not to let me see the operator's definition. I will tell you an optimization theorem that the operator satisfies. As an added bonus, I will also give you a proof of correctness for the optimisation, along with a formal guarantee about its effect on performance. The purpose of this paper is to explain these tricks.**

I. Introduction

At the risk of seeming off-message for a logic conference, modern programmers are too busy to bother proving theorems about their programs. *Free theorems* [1] provide a handy shortcut, as they allow us to conclude useful properties of functional programs without any proof whatsoever, provided the programs are suitably polymorphic. In this paper we apply this technique to the area of program optimisation, developing a generic optimisation pattern for recursive programs that can be instantiated for a wide variety of operators.

The focus of this paper is on a particular technique for optimising recursive programs: the *worker/wrapper transformation* [2]. This is a simple but general transformation based around a change of data representation. The transformation replaces a program that uses some type A with the combination of a *worker* program that uses a new type B and a *wrapper* program that converts between the two types as necessary. The idea behind the transformation is that the change of type allows the use of more efficient operations.

The worker/wrapper transformation was originally formulated and proved correct for recursive programs defined in terms of least-fixed points [2]. It has subsequently also been developed for programs written as folds [3] and unfolds [4], and some progress has been made towards unifying the various different approaches [5]. Most recently, in the context of least-fixed points it has been shown how rigorous arguments can be made about the effect of the transformation on a program's efficiency, based on improvement theory [6].

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

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

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

It is widely known that there is a relationship between (strong) dinaturality and parametricity, the property from which free theorems follow [7], [8]. The exact details of this relationship are unclear, and the situation is not helped by the wide variety of models of parametricity that have been developed. However, it is known that parametricity for certain types entails strong dinaturality [9]. For the purposes of this paper, we assume that all recursion operators of interest are strongly dinatural; in practice, we are not aware of any such operators in common use where this assumption fails.

Our paper has two main contributions. First of all, we develop a generic version of the worker/wrapper transformation, applicable to a wide class of recursion operators, with a correctness theorem based around the categorical notion of strong dinaturality. Secondly, we provide an equally general *improvement theorem* for the transformation, based around a modified notion of dinaturality we call "lax strong dinaturality". This theorem allows us to make formal guarantees about the performance effects of the worker/wrapper transformation when applied to the same wide class of operators. This guarantee can be phrased informally as "the worker/wrapper transformation never makes programs worse".

In this way, we establish strong dinaturality as the *essence* of the worker/wrapper transformation, and obtain a general theory that is applicable to a wide class of recursion operators. Furthermore, the efficiency side of the theory suggests a general categorical viewpoint on theories of program improvement in terms of *preorder-enriched categories*. This echoes earlier work by Hoare and others on a similar approach to program *refinement* [10], [11]. Finally, we observe that not only do all

existing worker/wrapper theories arise as instances of our new general theory, but the theory can also be used to derive new instances, including a theory for monadic fixed-points and an interesting degenerate case for arrow fixed-points.

II. Worker/Wrapper For Least-Fixed Points

In this section, we present a version of the worker/wrapper transformation for the particular case of recursive programs defined as least-fixed points, i.e. using general recursion, together with proofs of correctness. We then discuss the limitations of this presentation of the technique, and the problem of generalising to a wider range of recursion operators.

A. Review Of The Theory

The general form of the worker/wrapper transformation is based on a simple change of type. Given two types, A and B, along with functions $abs : B \to A$ and $rep : A \to B$, we attempt to factorise an original program that uses type A into a *worker* that uses the new type B and a *wrapper* that performs the necessary change of data representation. The wrapper allows the new worker program to be used in the same contexts as the original program. In order for this to be possible, we require the additional assumption that $abs \circ rep = id_A$, essentially capturing the idea that rep provides a *faithful* representation of elements of A in B. The worker/wrapper transformation can be presented with weaker forms of this assumption, but for now we only consider this strong formulation.

In the particular case for programs defined as least-fixed points, we have the original program written as a fixed point of a function $f : A \to A$, and wish to derive a new function $g : B \to B$, such that the following equation holds:

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

In this case, $fix\ f$ is the original program of type A, while abs is the wrapper and $fix\ g$ is the worker of type B.

Our use of least-fixed points means that we must choose an appropriate semantics basis where this notion makes sense; the traditional choice is the category $\mathbf{Cpo}$ of complete pointed partial orders and continuous functions. Our theory will also make use of the following two rules. Firstly, the *rolling* rule, allowing functions inside a fixed point to be "rolled" out:

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

Secondly, the *fusion* rule, which assuming the function h is strict (i.e. $h \perp = \perp$) can be stated as follows:

$$h\ (fix\ f) = fix\ g \quad \Leftarrow \quad h \circ f = g \circ h$$

The original paper [2] provided the following proof of correctness for the worker/wrapper transformation:

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

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

A later paper based on folds [3] gave a proof of correctness for the worker/wrapper transformation based on fusion. Adapting this proof to the fix case, we obtain:

$$\begin{array}{ll}
 & fix\ f = abs\ (fix\ g) \\
\Leftarrow & \{\ abs \circ rep = id\ \} \\
 & abs\ (rep\ (fix\ f)) = abs\ (fix\ g) \\
\Leftarrow & \{\ \text{unapplying } abs\ \} \\
 & rep\ (fix\ f) = fix\ g \\
\Leftarrow & \{\ \text{fusion, assuming } rep \text{ is strict}\ \} \\
 & rep \circ f = g \circ rep
\end{array}$$

This proof gives a *specification* for the new function g in terms of the given functions rep and f, from which the aim is then to calculate an implementation. It also appears as a subproof of the complete proof for the worker/wrapper transformation for fixed points that is presented in [5].

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

We illustrate this theory with a simple example. Consider the naïve polymorphic function $reverse : [T] \to [T]$ that reverses a list of elements of type T, defined by:

$$\begin{array}{ll}
reverse\ [\,] & = [\,] \\
reverse\ (x : xs) & = reverse\ xs + + [x]
\end{array}$$

This can be written as a least-fixed point as follows:

$$reverse = fix\ f$$

$$\begin{array}{ll}
f\ r\ [\,] & = [\,] \\
f\ r\ (x : xs) & = r\ xs + + [x]
\end{array}$$

To obtain a more efficient version of $reverse$ using the worker/wrapper transformation, we shall apply the idea of *difference lists*, lists represented by functions on lists. In particular, we take $A = [T] \to [T]$, $B = [T] \to [T] \to [T]$, and define $abs : B \to A$ and $rep : A \to B$ by:

$$abs\ func = \lambda xs \to func\ xs\ [\,]$$

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

It is straightforward to verify that $abs \circ rep = id$. Using the fusion-based formulation we take $rep \circ f = g \circ rep$ as our specification for the new function g, from which we calculate a definition by case analysis. Firstly, for the empty list:

$$\begin{array}{ll}
 & (rep \circ f)\ r\ [\,] \\
= & \{\ \text{composition}\ \} \\
 & rep\ (f\ r)\ [\,] \\
= & \{\ \text{definition of } rep\ \} \\
 & \lambda ys \to f\ r\ [\,] + + ys \\
= & \{\ \text{definition of } f\ \}
\end{array}$$

$$
\begin{array}{ll}
 & \lambda ys \to [\,] +\!\!+ \; ys \\
= & \quad \{ \; [\,] \text{ is the unit of } +\!\!+ \; \} \\
 & id \\
= & \quad \{ \text{ define } g \; r' \; [\,] = id \; \} \\
 & g \; (rep \; r) \; [\,] \\
= & \quad \{ \text{ composition } \} \\
 & (g \circ rep) \; r \; [\,]
\end{array}
$$

Secondly, for a non-empty list:

$$
\begin{array}{ll}
 & (rep \circ f) \; r \; (x : xs) \\
= & \quad \{ \text{ composition } \} \\
 & rep \; (f \; r) \; (x : xs) \\
= & \quad \{ \text{ definition of } rep \; \} \\
 & \lambda ys \to f \; r \; (x : xs) +\!\!+ \; ys \\
= & \quad \{ \text{ definition of } f \; \} \\
 & \lambda ys \to r \; xs +\!\!+ [x] +\!\!+ \; ys \\
= & \quad \{ \text{ definition of } +\!\!+ \; \} \\
 & \lambda ys \to r \; xs +\!\!+ (x : ys) \\
= & \quad \{ \text{ definition of } rep \; \} \\
 & \lambda ys \to rep \; r \; xs \; (x : ys) \\
= & \quad \{ \text{ define } g \; r' \; (x : xs) = \lambda ys \to r' \; xs \; (x : ys) \; \} \\
 & g \; (rep \; r) \; (x : xs) \\
= & \quad \{ \text{ composition } \} \\
 & (g \circ rep) \; r \; (x : xs)
\end{array}
$$

Note that the calculation for g did not require the use of induction. Expanding out the resulting new definition $reverse = abs \; (fix \; g)$, we obtain the familiar "fast reverse":

$$
\begin{array}{l}
reverse \; xs = rev' \; xs \; [\,] \\[4pt]
rev' \; [\,] \qquad\quad ys = ys \\
rev' \; (x : xs) \; ys = rev' \; xs \; (x : ys)
\end{array}
$$

Thus we see that the transformation from naïve to fast reverse is an instance of the worker/wrapper transformation.

B. Generalising From The Fix Case

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

Sculthorpe and Hutton [5] made some way toward generalising the worker/wrapper transformation, giving a uniform presentation of various worker/wrapper theories. However, this is somewhat unsatisfactory, as it does little to explain *why* such a uniform presentation is possible, only demonstrating that it is. Nevertheless, this work provided a vital stepping-stone toward the general theory we present in this paper.

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

III. Dinaturality and Strong Dinaturality

Now we shall explain the concepts of dinaturality and strong dinaturality, including their relationship with the rolling rule and fusion. For this section, we assume a small amount of knowledge of category theory, up to functors.

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

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

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

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

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

$$
\begin{array}{ccccc}
 & \mathsf{F}\;(A,A) & \xrightarrow{\;\alpha_A\;} & \mathsf{G}\;(A,A) & \\
{\scriptstyle \mathsf{F}\,(h,A)} \nearrow & & & & \searrow {\scriptstyle \mathsf{G}\,(A,h)} \\
\mathsf{F}\;(B,A) & & & & \mathsf{G}\;(A,B) \\
{\scriptstyle \mathsf{F}\,(B,h)} \searrow & & & & \nearrow {\scriptstyle \mathsf{G}\,(h,B)} \\
 & \mathsf{F}\;(B,B) & \xrightarrow[\;\alpha_B\;]{} & \mathsf{G}\;(B,B) &
\end{array}
$$

For *fix*, this property exactly captures the rolling rule. To see this, take $\mathcal{C} = \mathbf{Cpo}$ and $\mathcal{D} = \mathbf{Set}$ (the category of sets and total functions), let $\mathsf{F}\; (X, Y) = Hom\; (X, Y)$ and $\mathsf{G}\; (X, Y) = Y$, and assume we are given a continuous

function $f \in \mathsf{F}\ (B, A) = B \to A$. Then chasing the function f around the above diagram, we obtain the rolling rule:

$$
\begin{array}{ccccc}
 & f \circ h & \xmapsto{fix} & fix\ (f \circ h) & \\
{\scriptstyle Hom\ (h,A)} \nearrow & & & & \searrow {\scriptstyle h} \\
f & & & & h\ (fix\ (f \circ h)) = fix\ (h \circ f) \\
{\scriptstyle Hom\ (B,h)} \searrow & & & & \nearrow {\scriptstyle id} \\
 & h \circ f & \xmapsto[fix]{} & fix\ (h \circ f) &
\end{array}
$$

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

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

$$
\begin{array}{ccccccc}
 & \mathsf{F}\ (A, A) & \xrightarrow{\alpha_A} & \mathsf{G}\ (A, A) & & & \\
{\scriptstyle p} \nearrow & & \searrow {\scriptstyle \mathsf{F}\ (A,h)} & & \searrow {\scriptstyle \mathsf{G}\ (A,h)} & & \\
X & & \mathsf{F}\ (A, B) & \Rightarrow & \mathsf{G}\ (A, B) & & \\
{\scriptstyle q} \searrow & & \nearrow {\scriptstyle \mathsf{F}\ (h,B)} & & \nearrow {\scriptstyle \mathsf{G}\ (h,B)} & & \\
 & \mathsf{F}\ (B, B) & \xrightarrow[\alpha_B]{} & \mathsf{G}\ (B, B) & & &
\end{array}
$$

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

$$
\begin{array}{ccccc}
 & f & \xmapsto{fix} & fix\ f & \\
{\scriptstyle p} \nearrow & & \searrow {\scriptstyle Hom\ (A,h)} & & \searrow {\scriptstyle h} \\
x & & h \circ f = g \circ h & \Rightarrow & h\ (fix\ f) = fix\ g \\
{\scriptstyle q} \searrow & & \nearrow {\scriptstyle Hom\ (h,B)} & & \nearrow {\scriptstyle id} \\
 & g & \xmapsto[fix]{} & fix\ g &
\end{array}
$$

Thus, strong dinaturality in this case states that $h \circ f = g \circ h$ implies $h\ (fix\ f) = fix\ g$. The fusion rule is precisely this property, with an extra strictness condition on h. This condition can be recovered by treating F and G as functors from the *strict subcategory* $\mathbf{Cpo}_\perp$ in which all arrows are strict. The functor $\mathsf{F}\ (X, Y)$ is still defined as the full function space $\mathbf{Cpo}\ (X, Y)$, including non-strict arrows.

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

IV. WORKER/WRAPPER FOR STRONG DINATURALS

Suppose we have chosen a particular programming language to work with, and let $\mathcal{C}$ be the category where the objects are types in that language and the arrows are functions from one type to another. Then a polymorphic type $\forall x\ .\ T$ where x appears in both positive and negative positions in T can be represented by a functor $\mathsf{F} : \mathcal{C}^{op} \times \mathcal{C} \to \mathbf{Set}$, where $\mathsf{F}\ (A, A)$ is the set of terms in the language of type $T[A/x]$. In turn, a *recursion operator* that takes terms of type $\mathsf{F}\ (A, A)$ and produces terms of type $\mathsf{G}\ (A, A)$ can be represented by a strong dinatural transformation from F to G. It is known that for certain types, strong dinaturality will follow from a free theorem [9]. For example, the free theorem for the typing $fix : (A \to A) \to A$ is fusion, which we showed to be equivalent to strong dinaturality in the previous section.

Now we present the first of the two central results of this paper, in the form of a general categorical worker/wrapper theorem, which is summarised in Fig 1. The data in the theorem can be interpreted as follows:

- The category $\mathcal{C}$ is a programming language.
- The objects A and B are types in the language.
- The functors F, G are families of types.
- The arrows abs and rep are functions in the language.
- The elements f and g are terms in the language.
- The strong dinatural α is a recursion operator.

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

The wealth of conditions in Fig 1 requires some explanation. Previous worker/wrapper theorems in the literature had varying numbers of possible correctness conditions, ranging from just one in the original theory [2] to a total of five in [5]. This variation is a result of the way previous theories were first developed separately and then unified, and all previous conditions are included in some generalised form in our presentation. The nine conditions given in this paper were chosen to best expose the symmetries in the theory. In practical applications, one selects the condition that results in the simplest calculation for the worker program.

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

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

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

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

Given:

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

If one of the following conditions holds:

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

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

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

then we have the factorisation:

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

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

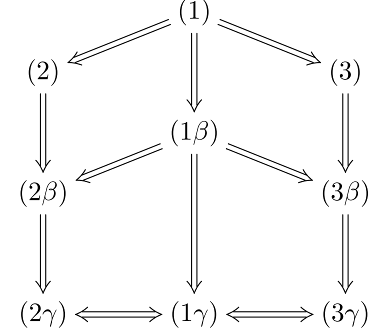


Fig. 1. The Worker/Wrapper Theorem for Strong Dinatural Transformations

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

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

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

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

The proof for (1γ) and (3γ) is dual. Thus we see that all three are equivalent. The basic relationships between the conditions are summarised in the right-hand side of Fig 1.

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

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

Conditions (1), (2) and (3) are precisely the three correctness conditions given in the original worker/wrapper theory for fold [3], while the corresponding β and γ conditions are weakenings of those conditions. The β conditions are simply the weakenings obtained by adding a recursive context, while the γ conditions are weakened further so that they are all equivalent, much like the two weakened conditions of Sculthorpe and Hutton [5]. However, those two conditions correspond here to the conditions (1β) and (2β), which in this generalised setting are not in general equivalent.

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

In earlier work, weaker versions of the assumption $abs \circ rep = id$ were also considered [2], [5]. For example, the proof of correctness for the original presentation of the

worker/wrapper transformation in terms of least-fixed points still holds if the assumption is weakened to $abs \circ rep \circ f = f$, or further to $fix\ (abs \circ rep \circ f) = fix\ f$.

The lack of weaker alternative assumptions means that our new theory is not a full generalisation of the earlier work. While this is not a significant issue, it is a little unsatisfactory. In our theorem, the assumption $abs \circ rep = id$ is used four times. For each of those four uses, a different weakening can be made. The four weakened versions are as follows:

(C1) $\alpha_A\ (\mathsf{F}\ (abs \circ rep, id)\ f) = \alpha_A\ f$
(C2) $\alpha_A\ (\mathsf{F}\ (id, abs \circ rep)\ f) = \alpha_A\ f$
(C3) $\mathsf{G}\ (id, abs \circ rep)\ (\alpha_A\ f) = \alpha_A\ f$
(C4) $\mathsf{G}\ (abs \circ rep, id)\ (\alpha_A\ f) = \alpha_A\ f$

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

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

(C1) $\alpha_A \circ \mathsf{F}\ (abs \circ rep, id) = \alpha_A$
(C2) $\alpha_A \circ \mathsf{F}\ (id, abs \circ rep) = \alpha_A$
(C3) $\mathsf{G}\ (id, abs \circ rep) \circ \alpha_A = \alpha_A$
(C4) $\mathsf{G}\ (abs \circ rep, id) \circ \alpha_A = \alpha_A$

in which case the duality holds exactly.

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

V. Examples

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

A. Least-Fixed Points

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

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

Recalling the discussion from section III, we note that the functors must be typed $\mathsf{F}, \mathsf{G} : \mathbf{Cpo}_\perp^{op} \times \mathbf{Cpo}_\perp \to \mathbf{Set}$ in order to obtain the correct strong dinaturality property.

By instantiating the theorem from Fig 1 for the fix operator, we obtain the following set of preconditions:

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

(1β) $fix\ g = fix\ (rep \circ f \circ abs)$
(2β) $fix\ g = rep\ (fix\ f)$
(3β) $abs\ (fix\ g) = fix\ f$

(1γ) $abs\ (fix\ g) = abs\ (fix\ (rep \circ f \circ abs))$
(2γ) $abs\ (fix\ g) = abs\ (rep\ (fix\ f)$
(3γ) $abs\ (fix\ g) = fix\ f$

Note that the functions abs and rep must be strict, because they are arrows in $\mathbf{Cpo}_\perp$. However, the hom-sets $\mathbf{Cpo}\ (A, A)$ and $\mathbf{Cpo}\ (B, B)$ are the *full* function spaces, so their respective elements f and g need not be strict. By instantiating the conclusion of the theorem we obtain the worker/wrapper factorisation $fix\ f = abs\ (fix\ g)$ from [2], [5].

As one would expect from the previous section, the first five of the preconditions correspond to the five conditions given for the general fix theory of Sculthorpe and Hutton [5]. However that theory had only one strictness requirement: for condition (2), rep must be strict to imply the conclusion. Here, we require both abs and rep to be strict for all conditions.

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

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

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

B. Folds

Next, we consider the fold operator. For a functor H with an *initial algebra* $\mu\mathsf{H}$, the fold operator for the type A takes an arrow of type $\mathsf{H}\ A \to A$ and extends it to an arrow of type $\mu\mathsf{H} \to A$. That is, we have the following typing:

$$fold : (\mathsf{H}\ A \to A) \to \mu\mathsf{H} \to A$$

The type $\mu\mathsf{H}$ can be thought of as a *least-fixed point* of H, and is a *canonical solution* to the equation $\mu\mathsf{H} \cong \mathsf{H}\ (\mu\mathsf{H})$. Informally, the fold operator reduces a structure of type $\mu\mathsf{H}$ to a single value of type A by recursing on all the subterms, and then assembling the subresults according to f.

One of the key properties of the fold operator is the *fusion* law. Given arrows $f : \mathsf{H}\ A \to A, g : \mathsf{H}\ B \to B, h : A \to B$, fusion is captured by the following implication:

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

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

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

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

Instantiating the worker/wrapper theorem from Fig 1 in this context gives the following set of preconditions:

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

(1β) $\mathit{fold}\ g = \mathit{fold}\ (\mathit{rep} \circ f \circ \mathsf{H}\ \mathit{abs})$
(2β) $\mathit{fold}\ g = \mathit{rep} \circ \mathit{fold}\ f$
(3β) $\mathit{abs} \circ \mathit{fold}\ g = \mathit{fold}\ f$

(1γ) $\mathit{abs} \circ \mathit{fold}\ g = \mathit{abs} \circ \mathit{fold}\ (\mathit{rep} \circ f \circ \mathsf{H}\ \mathit{abs})$
(2γ) $\mathit{abs} \circ \mathit{fold}\ g = \mathit{abs} \circ \mathit{rep} \circ \mathit{fold}\ f$
(3γ) $\mathit{abs} \circ \mathit{fold}\ g = \mathit{fold}\ f$

Instantiating the conclusion gives us $\mathit{fold}\ f = \mathit{abs} \circ \mathit{fold}\ g$. In this case, the first five preconditions and the conclusion are precisely those given for the fold theory in [5].

We note that it is unnecessary to assume anything about the object $\mu\mathsf{H}$ in this presentation: strong dinaturality is sufficient to get all the necessary properties of the fold operator. We speculate that this may be related to the link between strong dinaturality and initial algebras as observed by Uustalu [8].

C. Monadic Fixed Points

Monads are a mathematical construct commonly used in programming language theory to deal with effects such as state and exceptions [12]. Languages like Haskell use monads to embed effectful computations into a pure language. In this context, a value of type $\mathsf{M}\ A$ for some monad M is an *effectful* computation that produces a result of type A, where the nature of the underlying effect is captured by the monad M.

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

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

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

$$\mathit{xm} \gg\!= \mathit{return} = \mathit{xm}$$
$$\mathit{return}\ x \gg\!= f = f\ x$$
$$(\mathit{xm} \gg\!= f) \gg\!= g = \mathit{xm} \gg\!= (\lambda x \to f\ x \gg\!= g)$$

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

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

Sometimes it is useful to perform recursive computations within a monad. A normal fixed-point operator is usually insufficient for this, as depending on the implementation of the monad, it is easy to introduce nontermination. For this reason, some monads come equipped with a *monadic fix* operation $\mathit{mfix} : (A \to \mathsf{M}\ A) \to \mathsf{M}\ A$. Monadic fix operations are required to follow a number of laws, but here we concern ourselves only with one such law, which follows from parametricity [13]. For any strict function $s : A \to B$ and functions $f : A \to \mathsf{M}\ A$, $g : B \to \mathsf{M}\ B$, we have:

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

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

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

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

(1β) $\mathit{mfix}\ g = \mathit{mfix}\ (\mathsf{M}\ \mathit{rep} \circ f \circ \mathit{abs})$
(2β) $\mathit{mfix}\ g = \mathsf{M}\ \mathit{rep}\ (\mathit{mfix}\ f)$
(3β) $\mathsf{M}\ \mathit{abs}\ (\mathit{mfix}\ g) = \mathit{mfix}\ f$

(1γ) $\mathsf{M}\ \mathit{abs}\ (\mathit{mfix}\ g) = \mathsf{M}\ \mathit{abs}\ (\mathit{mfix}\ (\mathit{rep} \circ f \circ \mathit{abs}))$
(2γ) $\mathsf{M}\ \mathit{abs}\ (\mathit{mfix}\ g) = \mathsf{M}\ (\mathit{abs} \circ \mathit{rep})\ (\mathit{mfix}\ f)$
(3γ) $\mathsf{M}\ \mathit{abs}\ (\mathit{mfix}\ g) = \mathit{mfix}\ f$

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

D. Arrow Loops

Unfortunately, monads cannot capture all notions of effectful computation we may wish to use. For this reason, we may sometimes choose to use a more general framework such as

arrows [14]. An arrow is a binary type constructor Arr together with three operations of the following types:

$$\begin{aligned} arr &: (A \to B) \to \mathsf{Arr}\ A\ B \\ seq &: \mathsf{Arr}\ A\ B \to \mathsf{Arr}\ B\ \mathcal{C} \to \mathsf{Arr}\ A\ \mathcal{C} \\ first &: \mathsf{Arr}\ A\ B \to \mathsf{Arr}\ (A \times C)\ (B \times C) \end{aligned}$$

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

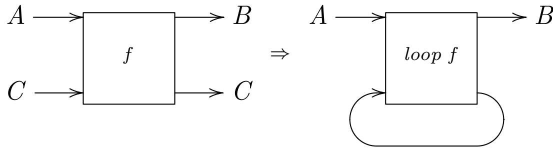
$$(f \ggg g) \ggg h = f \ggg (g \ggg h)$$

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

Thus far, arrows have no notion of recursion. However, some arrows provide an extra $loop$ combinator [15]:

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

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



Once again, loops are expected to satisfy a number of laws that we shall not list here. It follows from the laws of arrows and loops that if $f \ggg second\ (arr\ h) = second\ (arr\ h) \ggg g$ then $loop\ f = loop\ g$, implying that $loop$ is a dinatural transformation $\mathsf{F} \to \mathsf{G}$ between the following functors:

$$\begin{aligned} \mathsf{F}\ (X, Y) &= \mathsf{Arr}\ (A \times X)\ (B \times Y) \\ \mathsf{G}\ (X, Y) &= \mathsf{Arr}\ A\ B \end{aligned}$$

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

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

(1β) $loop\ g = loop\ (second\ (arr\ abs) \ggg f \ggg second\ (arr\ rep))$
(2β) $loop\ g = loop\ f$
(3β) $loop\ g = loop\ f$

(1γ) $loop\ g = loop\ (second\ (arr\ abs) \ggg f \ggg second\ (arr\ rep))$
(2γ) $loop\ g = loop\ f$
(3γ) $loop\ g = loop\ f$

then we can conclude $loop\ f = loop\ g$.

We have again instantiated our general theory to produce a novel worker/wrapper theory with very little effort, allowing the worker/wrapper transformation to be applied to programs written using the arrow loop combinator. Just as was the case for monadic recursion, our theorem is based entirely on the property of strong dinaturality, and thus does not require any of the arrow laws to hold beyond the assumption that the loop operator is strongly dinatural. Note that in this case we have a degenerate form of the conclusion where the wrapper is just the identity, because the functor G ignores its inputs.

VI. Worker/Wrapper For Improvement

Thus far, we have only addressed the problem of proving the generalised worker/wrapper transformation correct. However, any useful optimisation must do more than simply preserve the meaning of a program: the transformed program ought to be in some way better than the original with respect to resource usage. At the very least, it should not be worse. We refer to this as the problem of *improvement*.

In much work on program optimisation in the context of functional languages, discussion of improvement is limited to empirical measures such as benchmarks [16] and profiling [17]. This is because the operational behaviour of functional programs can be hard to predict, especially in call-by-need languages such as Haskell. In essence, empirical methods are used to circumvent the limitations of theory, which in the case of improvement is underdeveloped.

In this section, we develop a more rigorous approach to improvement for the worker/wrapper transformation. Firstly, we discuss *improvement theory* à la Sands, which formed the basis of the approach we used in our previous paper [6]. Secondly, we review the concept of preorder-enriched categories, the theoretical machinery we use to model improvement in a general categorical setting. Thirdly, we discuss the appropriate generalisation of strong dinaturality in this setting. Finally, we present a refined version of the worker/wrapper correctness theorem from Fig 1 that uses these ideas to formulate an improvement theorem, which can be used to verify efficiency properties of the transformation.

A. Improvement Theory à la Sands

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

This theory was developed initially by Sands for the particular case of the call-by-name lambda calculus [18]. Subsequently, Moran and Sands developed a theory for call-by-need time costs [19], while Gustavsson and Sands developed the corresponding theory for space usage [20], [21].

While improvement theory provides the machinery needed to reason about the behaviour of call-by-need languages that

are traditionally considered unpredictable, it is unfortunately limited by being tied to specific operational semantics. While subsequent work by Sands [22] goes some way toward rectifying this, we would like a general, categorical theory compatible with the approach we used earlier in this paper. For this reason, we shall use an alternate approach to reasoning about improvement based on preorder-enriched categories.

B. Improvement Theory Via Preorder-Enriched Categories

Category theory offers us one fundamental way to compare arrows: by asking if they are equal or not. This makes the theory ideal for reasoning about equivalence of programs. However, if we wish to reason about other properties, we require additional structure. For this purpose, we use the machinery of *enriched* category theory [23]. In general, categories can be enriched over a wide variety of structures, but in this case we shall use preorders to enrich our categories.

A *preorder-enriched category* is a category where each hom-set $Hom\ (A, B)$ is equipped with a preorder $\preceq$, and composition is monotonic with respect to this ordering:

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

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

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

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

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

The use of preorder-enriched categories to compare programs has previously been considered in the area of program *refinement* [10], [11]. While improvement is the problem of making a program more efficient, refinement is the related problem of making a program more *executable*, in the sense of transforming a specification into an implementation. Our focus is on improvement, but it is worth noting that all of the theory in this section can be applied equally to refinement.

C. Generalising Strong Dinaturality

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

$$X \xrightarrow{p} \mathsf{F}\,(A,A) \xrightarrow{\mathsf{F}\,(A,h)} \mathsf{F}\,(A,B), \quad X \xrightarrow{q} \mathsf{F}\,(B,B) \xrightarrow{\mathsf{F}\,(h,B)} \mathsf{F}\,(A,B) \quad \Rightarrow \quad \mathsf{F}\,(A,A) \xrightarrow{\alpha_A} \mathsf{G}\,(A,A) \xrightarrow{\mathsf{G}\,(A,h)} \mathsf{G}\,(A,B), \quad \mathsf{F}\,(B,B) \xrightarrow{\alpha_B} \mathsf{G}\,(B,B) \xrightarrow{\mathsf{G}\,(h,B)} \mathsf{G}\,(A,B)$$

Note that F and G are now functors between order-enriched categories that respect the arrow ordering. The diagram expresses the following implication in pictorial form:

$$\begin{array}{l} \quad \mathsf{F}\,(A,h) \circ p \preceq \mathsf{F}\,(h,B) \circ q \\ \Rightarrow \\ \quad \mathsf{G}\,(A,h) \circ \alpha_A \circ p \preceq \mathsf{G}\,(h,B) \circ \alpha_B \circ q \end{array}$$

We also use the term *oplax* when the ordering is reversed (i.e. using $\succeq$ rather than $\preceq$), and *bilax* when both lax and oplax properties hold. To rephrase, a bilax strong dinatural transformation must satisfy the above property for both the normal ordering on arrows and the inverse ordering. The choice of which direction is lax and which is oplax is arbitrary. For the purposes of this paper, we choose bilax strong dinaturality as our generalisation of strong dinaturality.

We specifically choose to use bilax strong dinaturality for two reasons. First of all, in our previous paper on improvement [6] we used a fusion theorem for fixed-points that bears a great deal of similarity to bilax strong dinaturality, and its bidirectionality was useful in proving the central theorem of that paper. Secondly, Johann and Voigtländer's technique to generate inequational free theorems from polymorphic types [24] results in precisely this same bidirectionality.

D. Worker/Wrapper Theorem For Improvement

By using bilax strong dinaturality as our generalisation of strong dinaturality, we can adapt the theorem we presented in Fig 1 to an *inequational* version. By exchanging **Set** in the theorem for the preorder-enriched category **Ord** of preorders and monotonic functions, we can make ordering comparisons between the two sides of each precondition and the conclusion. The resulting theorem is presented in Fig 2.

Note that we relax the assumption $abs \circ rep = id$ to $abs \circ rep \cong id$ in the new theorem, as the full strength of equality is no longer required. All other equalities have been weakened to inequalities. The resulting inequalities can be interpreted as comparisons of efficiency, where $f \preceq g$ means that f is *improved* by g in terms of efficiency, i.e. 'bigger' in the preorder means 'better' in efficiency. Under this interpretation, the theorem in Fig 2 gives efficiency conditions under which we can factorise an original program written as $\alpha_A\ f$ into a more efficient version comprising a worker program $\alpha_B\ g$ and a wrapper function $\mathsf{G}\,(rep, abs)$.

The proof of this theorem follows precisely the same form as the proof of the earlier theorem. The earlier proof can be

Given:

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

If one of the following conditions holds:

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

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

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

then we have the factorisation:

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

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

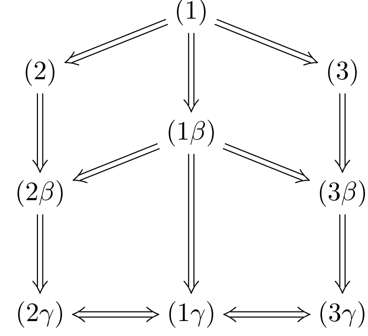


Fig. 2. The Worker/Wrapper Theorem for Bilax Strong Dinatural Transformations

transformed into a proof of this theorem by a straightforward process of replacing equalities with inequalities, so we shall refrain from giving the proof here.

This simple generalisation of our earlier theorem allows us to reason about *any* notion of improvement so long as we can convince ourselves that our recursion operator treats it parametrically. We conjecture that this will be the case for a wide class of resources and operators, though to prove this will likely require techniques similar to those used in [22].

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

E. Example: Least-Fixed Points

We can instantiate our theorem for least-fixed points once again. To do this, we take the same F and G as we did before, and simply change the target category from $\mathbf{Set}$ to $\mathbf{Ord}$, equipping the sets with the same ordering they had in the $\mathbf{Cpo}$ setting. This results in the following set of preconditions:

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

(1β) $\mathit{fix}\ g \succeq \mathit{fix}\ (rep \circ f \circ abs)$
(2β) $\mathit{fix}\ g \succeq rep\ (\mathit{fix}\ f)$
(3β) $abs\ (\mathit{fix}\ g) \succeq \mathit{fix}\ f$

(1γ) $abs\ (\mathit{fix}\ g) \succeq abs\ (\mathit{fix}\ (rep \circ f \circ abs))$
(2γ) $abs\ (\mathit{fix}\ g) \succeq abs\ (rep\ (\mathit{fix}\ f))$
(3γ) $abs\ (\mathit{fix}\ g) \succeq \mathit{fix}\ f$

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

The resulting improvement theorem for *fix* is similar to the version from our previous paper [6], with two differences. Firstly, our new theorem is for arbitrary resource usage in the $\mathbf{Cpo}$ setting, whereas the earlier theorem was specific to time performance. Secondly, the earlier theorem had no strictness conditions, whereas the above theorem does. In both cases, these differences are inherited from the fusion theorem or strong dinaturality property of the underlying theory.

F. Remarks

The process of generalising from the correctness theorem of Fig 1 to the improvement theorem of Fig 2 was entirely straightforward, our treatment of correctness leading immediately to a related treatment of improvement. This is encouraging, as it helps to justify our choice of machinery. Furthermore, the similarities between the two theorems mean

that it should be straightforward to adapt a proof of correctness into a proof of improvement, a benefit this work shares with our previous paper [6]. However, this work improves on the previous paper by showing that the correctness theorem can be considered a specialisation of the improvement theorem, which serves as progress toward uniting the two separate branches of correctness and efficiency. We hope that more work will go into unifying these aspects of program optimisation.

VII. Conclusion

We began this paper by stating that programmers are too busy to prove theorems, and the goal of this paper has been to reduce the number of theorems a programmer need prove. Our chosen mechanism for doing this was by developing the worker/wrapper transformation, a general-purpose program optimisation technique for recursive programs.

By utilising the categorical concept of strong dinaturality, we have developed a highly re-usable version of the worker/wrapper transformation whose correctness theorem can be instantiated for a wide class of recursion operators without the need for any proofs. Furthermore, with little extra work, we have expanded this correctness theorem into a theorem that can deal with the other side of optimisation, that of *improvement*. We also demonstrated the utility of these theorems by instantiating them for a number of patterns of recursion, including some for which no version of the worker/wrapper theorem had been developed before.

In keeping with our goal of avoiding proofs, what proofs there were in this paper have all been short and straightforward. This reflects the often-stated property of categorical thinking: "with the right definitions, the proofs are trivial".

There is much precedence for free theorems being used to prove the correctness of optimisations. In particular, the correctness of shortcut fusion (also known as $foldr/build$ fusion) relies on a free theorem that comes from the rank-2 type of $build$ [25]. Furthermore, the correctness of stream fusion [26] makes a similar use of free theorems [27]. This paper builds on this approach by using parametricity to confirm not just correctness of a program optimisiation, but also improvement. We hope that this approach to dealing with improvement will be re-used and expanded on in future work.

The inspiration for using strong dinaturality as a generalisation of fusion came from work by Uustalu [8]. This idea expands on earlier work where dinaturality was used in its non-strengthened form in axiomatisations of fixed-point operators [28], [29]. In that work it served a similar role as in this paper, as an analogue of the rolling rule.

Beyond the technical results of this paper, there are two key ideas we hope readers will take away with them. The first idea is that making the right observations about the deep mathematical structure of a problem or theory can lead to straightforward generalisations. If we strip away all the categorical work, this paper boils down to the observation that the fusion and rolling rules are the essence of the worker/wrapper transformation. All the other work comes from simply trying to generalise these rules as far as they will go.

The second idea that we hope readers will take away is that higher category theory doesn't need to be complicated to be useful. While we have used enriched category theory in this paper to reason about program improvement, we have tried to make these ideas as accessible as possible. Despite this drive toward accessibility and simplicity, the results we have proved with this framework are decidedly non-trivial. In short, a little enriched category theory can go a long way.

There are a number of potential avenues for further work on these ideas. Probably the most straightforward next step would be to instantiate the new theory we have developed for various operators, and investigate the particular properties of each instantiation. This would have two benefits: firstly, it would make it easier for programmers to use this theory for their own applications, and secondly, the particular properties of each instantiation may suggest ways to develop the general theory. In particular, it would be interesting to see what other assumptions, if any, are needed to include weakened versions of the $abs \circ rep = id$ assumption as seen in earlier papers on the worker/wrapper transformation [2], [5], [6].

Another way to develop this work would be to investigate particular models of program equivalence and efficiency. By developing plausible models where our underlying assumptions hold, we can argue that our assumptions are justified in the general case. In particular, it would be useful to further investigate the relationship between parametricity and bilax strong dinaturality, hopefully developing a notion of parametricity which implies bilax strong dinaturality for all relevant types. If this should turn out to be impossible, we would also like to know why this is the case.

At the moment, in order to verify preconditions for our improvement theorem, one would need to fall back on a pre-existing theory of improvement based on the operational semantics of a programming language. This is undesirable, as it increases the amount of theoretical knowledge and proof skills that a programmer would need to use our theory. We would like to develop a richer theory of bilax strong dinaturality as applied to program improvement, to see if this assumption can be used elsewhere to prove improvement relations. Ultimately, we would like to see a purely categorical theory of improvement, allowing improvement relations to be proved in a purely abstract way without the need to reason at the level of an underlying concrete semantics.

Finally, we would like to investigate the potential for automating the worker/wrapper transformation. By far the biggest hurdle for this would be automating the verification of the preconditions. We believe that the best approach to doing this would be to adapt algorithms designed for *higher-order unification* [30], the problem of solving equations on lambda terms. It may even be possible to adapt these algorithms to deal with the *inequational* conditions of our improvement theorem.

Acknowledgments

The authors would like to thank Thorsten Altenkirch and Neil Sculthorpe for useful discussions regarding this work,

and Clarissa Littler and Philippa Cowderoy for their helpful comments on draft versions of the paper.

REFERENCES

[1] P. Wadler, "Theorems for Free!" in *Functional Programming Languages and Computer Architecture*, 1989.
[2] A. Gill and G. Hutton, "The Worker/Wrapper Transformation," *Journal of Functional Programming*, vol. 19, no. 2, 2009.
[3] G. Hutton, M. Jaskelioff, and A. Gill, "Factorising Folds for Faster Functions," *Journal of Functional Programming Special Issue on Generic Programming*, vol. 20(3&4), 2010.
[4] J. Hackett, G. Hutton, and M. Jaskelioff, "The Under Performing Unfold: A New Approach to Optimising Corecursive Programs," in *Symposium on Implementation and Application of Functional Languages*, 2013.
[5] N. Sculthorpe and G. Hutton, "Work It, Wrap It, Fix It, Fold It," *Journal of Functional Programming*, vol. 24, no. 1, 2014.
[6] J. Hackett and G. Hutton, "Worker/Wrapper/Makes It/Faster," in *International Conference on Functional Programming*, 2014.
[7] E. S. Bainbridge, P. J. Freyd, A. Scedrov, and P. J. Scott, "Functorial Polymorphism," *Theoretical Computer Science*, vol. 70, no. 1, 1990.
[8] T. Uustalu, "A Note on Strong Dinaturality, Initial Algebras and Uniform Parameterized Fixpoint Operators," in *Fixed Points in Computer Science*, 2010.
[9] N. Ghani, T. Uustalu, and V. Vene, "Build, Augment and Destroy, Universally," in *Programming Languages and Systems: Second Asian Symposium*, 2004.
[10] C. A. R. Hoare, "Data Refinement in a Categorical Setting," 1987, typed manuscript.
[11] R.-J. Back and J. Wright, *Refinement Calculus: A Systematic Introduction*. Springer, 1998.
[12] P. Wadler, "The Essence of Functional Programming," in *Principles of Programming Languages*, 1992.
[13] L. Erkök and J. Launchbury, "Recursive Monadic Bindings," in *International Conference on Functional Programming*, 2000.
[14] J. Hughes, "Generalising Monads to Arrows," *Science of Computer Programming*, vol. 37, no. 1-3, 2000.
[15] R. Paterson, "A New Notation for Arrows," in *International Conference on Functional Programming*, 2001.
[16] W. Partain, "The nofib Benchmark Suite of Haskell Programs," in *Glasgow Workshop on Functional Programming*, 1992.
[17] P. M. Sansom and S. L. Peyton Jones, "Formally Based Profiling for Higher-Order Functional Languages," *ACM Transactions on Programming Languages and Systems*, vol. 19, no. 2, 1997.
[18] D. Sands, "Operational Theories of Improvement in Functional Languages (Extended Abstract)," in *Glasgow Workshop on Functional Programming*, 1991.
[19] A. Moran and D. Sands, "Improvement in a Lazy Context: An Operational Theory for Call-by-Need," extended version of [31], available at http://tinyurl.com/ohuv8ox.
[20] J. Gustavsson and D. Sands, "A Foundation for Space-Safe Transformations of Call-by-Need Programs," *Electronic Notes on Theoretical Computer Science*, vol. 26, 1999.
[21] ——, "Possibilities and Limitations of Call-by-Need Space Improvement," in *International Conference on Functional Programming*, 2001.
[22] D. Sands, "From SOS Rules to Proof Principles: An Operational Metatheory for Functional Languages," in *Principles of Programming Languages*, 1997.
[23] G. M. Kelly, "Basic Concepts of Enriched Category Theory," in *LMS Lecture Notes*, vol. 64. Cambridge University Press, 1982.
[24] P. Johann and J. Voigtländer, "Free Theorems in the Presence of *seq*," in *Principles of Programming Languages*, 2004.
[25] A. J. Gill, J. Launchbury, and S. L. Peyton Jones, "A Short Cut to Deforestation," in *Functional Programming Languages and Computer Architecture*, 1993.
[26] D. Coutts, R. Leshchinskiy, and D. Stewart, "Stream Fusion: From Lists to Streams to Nothing at All," in *International Conference on Functional Programming*, 2007.
[27] D. Coutts, "Stream Fusion: Practical Shortcut Fusion for Coinductive Sequence Types," Ph.D. dissertation, University of Oxford, 2010.
[28] A. K. Simpson, "A Characterisation of the Least-Fixed-Point Operator by Dinaturality," *Theoretical Computer Science*, vol. 118, no. 2, 1993.
[29] A. K. Simpson and G. D. Plotkin, "Complete Axioms for Categorical Fixed-Point Operators," in *Logic in Computer Science*, 2000.
[30] G. P. Huet, "A Unification Algorithm for Typed λ-Calculus," *Theoretical Computer Science*, vol. 1, no. 1, 1975.
[31] A. Moran and D. Sands, "Improvement in a Lazy Context: An Operational Theory for Call-by-Need," in *Principles of Programming Languages*, 1999.