Mechanising the worker/wrapper transformation

Peter Gammie

April 20, 2020

Contents

1 Introduction 2

2 Fixed-point theorems for program transformation 2
  2.1 The rolling rule ........................................ 2
  2.2 Least-fixed-point fusion .................................. 3

3 The transformation according to Gill and Hutton 3
  3.1 Worker/wrapper fusion is partially correct .............. 5
  3.2 A non-strict unwrap may go awry .......................... 6

4 A totally-correct fusion rule 7

5 Naive reverse becomes accumulator-reverse. 8
  5.1 Hughes lists, naive reverse, worker-wrapper optimisation. . 8
  5.2 Gill/Hutton-style worker/wrapper. .......................... 10
  5.3 Optimise worker/wrapper. ................................. 10

6 Unboxing types. 12
  6.1 Factorial example. ........................................ 12
  6.2 Introducing an accumulator. .............................. 14

7 Memoisation using streams. 16
  7.1 Streams. .................................................. 16
  7.2 The wrapper/unwrapper functions. ......................... 17
  7.3 Fibonacci example. ...................................... 17

8 Tagless interpreter via double-barreled continuations 19
  8.1 Worker/wrapper ........................................... 19

9 Backtracking using lazy lists and continuations 21
### 1 Introduction

This mechanisation of the worker/wrapper theory of Gill and Hutton (2009) was carried out in Isabelle/HOLCF (Müller et al. 1999; Huffman 2009). It accompanies Gammie (2011). The reader should note that oo stands for function composition, Λ for continuous function abstraction, · for continuous function application, domain for recursive-datatype definition.  

### 2 Fixed-point theorems for program transformation

We begin by recounting some standard theorems from the early days of denotational semantics. The origins of these results are lost to history; the interested reader can find some of it in Bekić (1984); Manna (1974); Greibach (1975); Stoy (1977); de Bakker et al. (1980); Harel (1980); Plotkin (1983); Winskel (1993); Sangiorgi (2009).

#### 2.1 The rolling rule

The rolling rule captures what intuitively happens when we re-order a recursive computation consisting of two parts. This theorem dates from the 1970s at the latest – see Stoy (1977, p210) and Plotkin (1983). The following proofs were provided by Gill and Hutton (2009).

**lemma** rolling-rule-ltr: \( \text{fix} \cdot (g \circ f) \subseteq g \cdot (\text{fix} \cdot (f \circ g)) \)

(\text{proof})

**lemma** rolling-rule-rtl: \( g \cdot (\text{fix} \cdot (f \circ g)) \subseteq \text{fix} \cdot (g \circ f) \)

(\text{proof})

**lemma** rolling-rule: \( \text{fix} \cdot (g \circ f) = g \cdot (\text{fix} \cdot (f \circ g)) \)

(\text{proof})
2.2 Least-fixed-point fusion

Least-fixed-point fusion provides a kind of induction that has proven to be very useful in calculational settings. Intuitively it lifts the step-by-step correspondence between \( f \) and \( h \) witnessed by the strict function \( g \) to the fixed points of \( f \) and \( g \):

\[
\begin{array}{cccc}
\bullet & \xrightarrow{h} & \bullet \\
g & \parallel & g \\
\bullet & \xrightarrow{f} & \bullet \\
\end{array}
\implies
\begin{array}{cccc}
\bullet & \xrightarrow{\text{fix } h} & \bullet \\
g & \parallel & g \\
\bullet & \xrightarrow{\text{fix } f} & \bullet \\
\end{array}
\]

Fokkinga and Meijer (1991), and also their later Meijer, Fokkinga, and Paterson (1991), made extensive use of this rule, as did Tullsen (2002) in his program transformation tool PATH. This diagram is strongly reminiscent of the simulations used to establish refinement relations between imperative programs and their specifications (de Roever and Engelhardt 1998).

The following proof is close to the third variant of Stoy (1977, p215). We relate the two fixpoints using the rule parallel_fix_ind:

\[
\begin{align*}
\text{adm } (\lambda x. \ ?P \ (\text{fst } x) \ (\text{snd } x)) \\
\ ?P \bot \bot \\
\land x \ y. \ ?P \ x \ y \\
\ ?P \ (\text{fix } \cdot \ ?F) \ (\text{fix } \cdot \ ?G)
\end{align*}
\]

in a very straightforward way:

**lemma lfp-fusion:**

assumes \( g \cdot \bot = \bot \)

assumes \( g \circ f = h \circ g \)

shows \( g \cdot (\text{fix } f) = \text{fix } h \)

**proof**

This lemma also goes by the name of Plotkin’s axiom (Pitts 1996) or uniformity (Simpson and Plotkin 2000).

**proof** **proof** **proof** **proof** **proof**

3 The transformation according to Gill and Hutton

The worker/wrapper transformation and associated fusion rule as formalised by Gill and Hutton (2009) are reproduced in Figure 1, and the reader is referred to the original paper for further motivation and background.

Armed with the rolling rule we can show that Gill and Hutton’s justification of the worker/wrapper transformation is sound. There is a battery of these transformations with varying strengths of hypothesis.
For a recursive definition $\text{comp} = \text{fix} \cdot \text{body}$ for some $\text{body} :: A \to A$ and a pair of functions $\text{wrap} :: B \to A$ and $\text{unwrap} :: A \to B$ where $\text{wrap} \circ \text{unwrap} = \text{id}_A$, we have:

$$
\begin{align*}
\text{comp} &= \text{wrap} \cdot \text{work} \\
\text{work} :: B \\
\text{work} &= \text{fix} \cdot (\text{unwrap} \circ \text{body} \circ \text{wrap})
\end{align*}
$$

(the worker/wrapper transformation)

Also:

$$(\text{unwrap} \circ \text{wrap}) \cdot \text{work} = \text{work} \quad \text{(worker/wrapper fusion)}$$

Figure 1: The worker/wrapper transformation and fusion rule of Gill and Hutton (2009).

The first requires $\text{unwrap} \circ \text{wrap}$ to be the identity for all values.

**Lemma worker-wrapper-id:**

- **Fixes** $\text{wrap} :: \forall a. \text{pcpo} \to \forall a. \text{pcpo}$
- **Fixes** $\text{unwrap} :: \forall a. \text{pcpo} \to \forall a. \text{pcpo}$
- **Assumes** $\text{wrap} \circ \text{unwrap} = \text{id}$
- **Assumes** $\text{comp} \circ \text{body} :: \text{computation} = \text{fix} \circ \text{body}$
- **Shows** $\text{computation} = \text{wrap} \circ \text{fix} \circ (\text{unwrap} \circ \text{body} \circ \text{wrap})$

**Proof**

The second weakens this assumption by requiring that $\text{wrap} \circ \text{wrap}$ only act as the identity on values in the image of $\text{body}$.

**Lemma worker-wrapper-body:**

- **Fixes** $\text{wrap} :: \forall a. \text{pcpo} \to \forall a. \text{pcpo}$
- **Fixes** $\text{unwrap} :: \forall a. \text{pcpo} \to \forall a. \text{pcpo}$
- **Assumes** $\text{wrap} \circ \text{unwrap} = \text{body}$
- **Assumes** $\text{comp} \circ \text{body} :: \text{computation} = \text{fix} \circ \text{body}$
- **Shows** $\text{computation} = \text{wrap} \circ \text{fix} \circ (\text{unwrap} \circ \text{body} \circ \text{wrap})$

**Proof**

This is particularly useful when the computation being transformed is strict in its argument.

Finally we can allow the identity to take the full recursive context into account. This rule was described by Gill and Hutton but not used.

**Lemma worker-wrapper-fix:**

- **Fixes** $\text{wrap} :: \forall a. \text{pcpo} \to \forall a. \text{pcpo}$
- **Fixes** $\text{unwrap} :: \forall a. \text{pcpo} \to \forall a. \text{pcpo}$
- **Assumes** $\text{wrap} \circ \text{unwrap} = \text{fix} \circ (\text{unwrap} \circ \text{body} \circ \text{unwrap}) \circ \text{body}$
- **Assumes** $\text{comp} \circ \text{body} :: \text{computation} = \text{fix} \circ \text{body}$
- **Shows** $\text{computation} = \text{wrap} \circ \text{fix} \circ (\text{unwrap} \circ \text{body} \circ \text{unwrap})$
Gill and Hutton’s worker-wrapper-fusion rule is intended to allow the transformation of \((\text{unwrap} \circ \text{wrap}) \cdot R\) to \(R\) in recursive contexts, where \(R\) is meant to be a self-call. Note that it assumes that the first worker/wrapper hypothesis can be established.

**lemma** worker-wrapper-fusion:

- **fixes** \(\text{wrap} :: 'b::\text{pcpo} \rightarrow 'a::\text{pcpo}\)
- **fixes** \(\text{unwrap} :: 'a \rightarrow 'b\)
- **assumes** \(\text{wrap-unwrap}: \text{wrap} \circ \text{unwrap} = \text{ID}\)
- **assumes** \(\text{work}: \text{work} = \text{fix}(\text{unwrap} \circ \text{body} \circ \text{wrap})\)
- **shows** \((\text{unwrap} \circ \text{wrap}) \cdot \text{work} = \text{work}\)

The following sections show that this rule only preserves partial correctness. This is because Gill and Hutton apply it in the context of the fold/unfold program transformation framework of Burstall and Darlington (1977), which need not preserve termination. We show that the fusion rule does in fact require extra conditions to be totally correct and propose one such sufficient condition.

### 3.1 Worker/wrapper fusion is partially correct

We now examine how Gill and Hutton apply their worker/wrapper fusion rule in the context of the fold/unfold framework.

The key step of those left implicit in the original paper is the use of the fold rule to justify replacing the worker with the fused version. Schematically, the fold/unfold framework maintains a history of all definitions that have appeared during transformation, and the fold rule treats this as a set of rewrite rules oriented right-to-left. (The unfold rule treats the current working set of definitions as rewrite rules oriented left-to-right.) Hence as each definition \(f = \text{body}\) yields a rule of the form \(\text{body} \Rightarrow f\), one can always derive \(f = f\). Clearly this has dire implications for the preservation of termination behaviour.

Tullsen (2002) in his §3.1.2 observes that the semantic essence of the fold rule is Park induction:

\[
\begin{align*}
\frac{f \cdot ?x = ?x}{\text{fix}_f \subseteq ?x} & \quad \text{fix}_\text{least}
\end{align*}
\]

viz that \(f x = x\) implies only the partially correct \(\text{fix} f \subseteq x\), and not the totally correct \(\text{fix} f = x\). We use this characterisation to show that if \(\text{unwrap}\) is non-strict (i.e. \(\text{unwrap} \parallel \neq \parallel\)) then there are programs where worker/wrapper fusion as used by Gill and Hutton need only be partially correct.
Consider the scenario described in Figure 1. After applying the worker/wrapper transformation, we attempt to apply fusion by finding a residual expression $\textit{body}^\prime$ such that the body of the worker, i.e. the expression $\textit{unwrap} \circ \textit{body} \circ \textit{wrap}$, can be rewritten as $\textit{body}^\prime \circ \textit{unwrap} \circ \textit{wrap}$. Intuitively this is the semantic form of workers where all self-calls are fusible. Our goal is to justify redefining $\textit{work}$ to $\text{fix} \cdot \textit{body}^\prime$, i.e. to establish:

$$\text{fix} \cdot (\textit{unwrap} \circ \textit{body} \circ \textit{wrap}) = \text{fix} \cdot \textit{body}^\prime$$

We show that worker/wrapper fusion as proposed by Gill and Hutton is partially correct using Park induction:

\begin{itemize}
    \item \textbf{lemma} fusion-partially-correct:
        \begin{itemize}
            \item \textbf{assumes} wrap-unwrap: $\textit{wrap} \circ \textit{unwrap} = \text{ID}$
            \item \textbf{assumes} work: $\textit{work} = \text{fix} \cdot (\textit{unwrap} \circ \textit{body} \circ \textit{wrap})$
            \item \textbf{assumes} body': $\textit{unwrap} \circ \textit{body} \circ \textit{wrap} = \textit{body}^\prime \circ \textit{unwrap} \circ \textit{wrap}$
            \item \textbf{shows} $\text{fix} \cdot \textit{body}^\prime \subseteq \textit{work}$
        \end{itemize}
\end{itemize}

The next section shows the converse does not obtain.

### 3.2 A non-strict \textit{unwrap} may go awry

If $\textit{unwrap}$ is non-strict, then it is possible that the fusion rule proposed by Gill and Hutton does not preserve termination. To show this we take a small artificial example. The type $A$ is not important, but we need access to a non-bottom inhabitant. The target type $B$ is the non-strict lift of $A$.

\begin{itemize}
    \item \textbf{domain} $A = A$
    \item \textbf{domain} $B = B \ (\text{lazy} \ A)$
\end{itemize}

The functions $\textit{wrap}$ and $\textit{unwrap}$ that map between these types are routine. Note that $\textit{wrap}$ is (necessarily) strict due to the property $\forall x. \ ?f \cdot (?g \cdot x) = x \implies ?f \cdot \bot = \bot$.

\begin{itemize}
    \item \textbf{fixrec} \textit{wrap} :: $B \to A$
        \begin{itemize}
            \item \textbf{where} $\textit{wrap} \cdot (B \cdot a) = a$
            \item \textbf{proof}
        \end{itemize}
    \item \textbf{fixrec} \textit{unwrap} :: $A \to B$
        \begin{itemize}
            \item \textbf{where} $\textit{unwrap} = B$
        \end{itemize}
\end{itemize}

Discharging the worker/wrapper hypothesis is similarly routine.

\begin{itemize}
    \item \textbf{lemma} wrap-unwrap: $\textit{wrap} \circ \textit{unwrap} = \text{ID}$
        \begin{itemize}
            \item \textbf{proof}
        \end{itemize}
\end{itemize}

The candidate computation we transform can be any that uses the recursion parameter $r$ non-strictly. The following is especially trivial.

\begin{itemize}
    \item \textbf{fixrec} \textit{body} :: $A \to A$
        \begin{itemize}
            \item \textbf{where} $\textit{body} \cdot r = A$
        \end{itemize}
\end{itemize}
The wrinkle is that the transformed worker can be strict in the recursion parameter \( r \), as \textit{unwrap} always lifts it.

\[
\text{fixrec } \textit{body}' :: B \to B \\
\text{where } \textit{body}' \cdot (B \cdot a) = B \cdot A(\text{proof})
\]

As explained above, we set up the fusion opportunity:

\[
\text{lemma } \langle \text{proof} \rangle
\quad \text{body-body': unwrap oo body oo wrap = body' oo unwrap oo wrap}
\]

This result depends crucially on \textit{unwrap} being non-strict.

Our earlier result shows that the proposed transformation is partially correct:

\[
\text{lemma } \langle \text{proof} \rangle
\quad \text{fix-body' } \subseteq \text{fix-(unwrap oo body oo wrap)}
\]

However it is easy to see that it is not totally correct:

\[
\text{lemma } \langle \text{proof} \rangle
\quad \lnot \text{fix-(unwrap oo body oo wrap) } \subseteq \text{fix-body'}
\]

This trick works whenever \textit{unwrap} is not strict. In the following section we show that requiring \textit{unwrap} to be strict leads to a straightforward proof of total correctness.

Note that if we have already established that \( \text{wrap oo unwrap = ID} \), then making \textit{unwrap} strict preserves this equation:

\[
\text{lemma } \langle \text{proof} \rangle
\quad \text{assumes wrap oo unwrap = ID} \\
\text{shows wrap oo strictify-unwrap = ID}
\]

From this we conclude that the worker/wrapper transformation itself cannot exploit any laziness in \textit{unwrap} under the context-insensitive assumptions of \textit{worker-wraper-id}. This is not to say that other program transformations may not be able to.

\[
\text{⟨proof}⟩
\]

\section{4 A totally-correct fusion rule}

We now show that a termination-preserving worker/wrapper fusion rule can be obtained by requiring \textit{unwrap} to be strict. (As we observed earlier, \textit{wrap} must always be strict due to the assumption that \( \text{wrap oo unwrap = ID} \).)

Our first result shows that a combined worker/wrapper transformation and fusion rule is sound, using the assumptions of \textit{worker-wraper-id} and the ubiquitous \textit{lfp-fusion} rule.

\[
\text{lemma } \langle \text{proof} \rangle
\quad \text{worker-wraper-fusion-new:}
\]
For a recursive definition \( \text{comp} = \text{body} \) of type \( A \) and a pair of functions \( \text{wrap} :: B \rightarrow A \) and \( \text{unwrap} :: A \rightarrow B \) where \( \text{wrap} \circ \text{unwrap} = \text{id}_A \) and \( \text{unwrap} \perp = \perp \), define:

\[
\begin{align*}
\text{comp} &= \text{wrap} \text{work} \\
\text{work} &= \text{unwrap} (\text{body}[\text{wrap} \text{work} / \text{comp}]) \\
\end{align*}
\]

(the worker/wrapper transformation)

In the scope of \( \text{work} \), the following rewrite is admissible:

\[
\text{unwrap} (\text{wrap} \text{work}) \implies \text{work}
\]

(worker/wrapper fusion)

Figure 2: The syntactic worker/wrapper transformation and fusion rule.

\[
\begin{align*}
\text{fixes} \ 	ext{wrap} :: 'b::\text{pcpo} \rightarrow 'a::\text{pcpo} \\
\text{fixes} \ 	ext{unwrap} :: 'a \rightarrow 'b \\
\text{fixes} \ 	ext{body}' :: 'b \rightarrow 'b \\
\text{assumes} \ \text{wrap-unwrap}: \text{wrap} \circ \text{unwrap} = \text{id}_A' \\
\text{assumes} \ \text{unwrap-strict}: \text{unwrap} \perp = \perp \\
\text{assumes} \ \text{body-body}': \text{unwrap} \circ \text{body} \circ \text{unwrap} = \text{body}' (\text{unwrap} \circ \text{unwrap}) \\
\text{shows} \ \text{fix-body} = \text{wrap} (\text{fix-body}') \\
\end{align*}
\]

We can also show a more general result which allows fusion to be optionally performed on a per-recursive-call basis using parallel fix ind:

\[
\begin{align*}
\text{lemma} \ \text{worker-wrapper-fusion-new-general}: \\
\text{fixes} \ 	ext{wrap} :: 'b::\text{pcpo} \rightarrow 'a::\text{pcpo} \\
\text{fixes} \ 	ext{unwrap} :: 'a \rightarrow 'b \\
\text{assumes} \ \text{wrap-unwrap}: \text{wrap} \circ \text{unwrap} = \text{id}_A' \\
\text{assumes} \ \text{unwrap-strict}: \text{unwrap} \perp = \perp \\
\text{assumes} \ \text{body-body}': \ \forall r. \ (\text{unwrap} \circ \text{unwrap})r \circ \text{body} = \text{body}'r \\
\text{shows} \ \text{fix-body} = \text{wrap} (\text{fix-body}') \\
\end{align*}
\]

This justifies the syntactically-oriented rules shown in Figure 2; note the scoping of the fusion rule.

Those familiar with the “bananas” work of Meijer, Fokkinga, and Paterson (1991) will not be surprised that adding a strictness assumption justifies an equational fusion rule.
5 Naive reverse becomes accumulator-reverse.

5.1 Hughes lists, naive reverse, worker-wrapper optimisation.

The “Hughes” list type.

type-synonym ‘a H = ‘a llist → ‘a llist

definition list2H :: ‘a llist → ‘a H where
  list2H ≡ lappend

lemma acc-c2a-strict[simp]: list2H · ⊥ = ⊥

(proof)

definition H2list :: ‘a H → ‘a llist where
  H2list = Λ f . f·lnil

The paper only claims the homomorphism holds for finite lists, but in fact it holds for all lazy lists in HOLCF. They are trying to dodge an explicit appeal to the equation ⊥ = (Λ x. ⊥), which does not hold in Haskell.

lemma H-list-hom-append: list2H·(xs ++ ys) = list2H·xs oo list2H·ys (is ?lhs
  = ?rhs)

(proof)

lemma H-list-hom-id: list2H·lnil = ID

(proof)

lemma H2list-list2H-inv: H2list oo list2H = ID

(proof)

Gill and Hutton (2009, §4.2) define the naive reverse function as follows.

fixrec lrev :: ‘a llist → ‘a llist

where
  lrev·lnil = lnil
  | lrev·(x :@ xs) = lrev·xs ++ (x :@ lnil)

Note “body” is the generator of lrev-def.

lemma lrev-strict[simp]: lrev·⊥ = ⊥

(proof)

fixrec lrev-body :: (‘a llist → ‘a llist) → ‘a llist → ‘a llist

where
  lrev-body·r·lnil = lnil
  | lrev-body·r·(x :@ xs) = r·xs ++ (x :@ lnil)

lemma lrev-body-strict[simp]: lrev-body·r·⊥ = ⊥
This is trivial but syntactically a bit touchy. Would be nicer to define \( \text{lrev-body} \) as the generator of the fixpoint definition of \( \text{lrev} \) directly.

**Lemma** \( \text{lrev-lrev-body-eq} \): \( \text{lrev} = \text{fix\text{-}lrev-body} \)

Wrap / unwrap functions.

**Definition**
\[ \text{unwrapH} :: ('a llist \to 'a llist) \to 'a llist \to 'a H \]
\[ \text{unwrapH} \equiv \Lambda f \cdot \text{list2H} \cdot (f \cdot x) \]

**Lemma** \( \text{unwrapH\text{-}strict[simp]} \): \( \text{unwrapH} \cdot \bot = \bot \)

**5.2 Gill/Hutton-style worker/wrapper.**

**Definition**
\[ \text{lrev-work} :: 'a llist \to 'a H \]
\[ \text{lrev-work} \equiv \text{fix} \cdot (\text{unwrapH} \circ \text{lrev-body} \circ \text{wrapH}) \]

**Definition**
\[ \text{lrev-wrap} :: 'a llist \to 'a llist \]
\[ \text{lrev-wrap} \equiv \text{wrapH} \cdot \text{lrev-work} \]

**Lemma** \( \text{lrev-lrev\text{-}ww-eq} \): \( \text{lrev} = \text{lrev-wrap} \)

**5.3 Optimise worker/wrapper.**

Intermediate worker.

**Fixrec** \( \text{lrev-body1} :: ('a llist \to 'a H) \to 'a llist \to 'a H \)

**Definition**
\[ \text{lrev-work1} :: 'a llist \to 'a H \]
\[ \text{lrev-work1} \equiv \text{fix}\text{-}lrev-body1 \]

**Lemma** \( \text{lrev-body\text{-}lrev-body1-eq} \): \( \text{lrev-body1} = \text{unwrapH} \circ \text{lrev-body} \circ \text{wrapH} \)
proof

lemma \textit{lrev-work1-lrev-work-eq}: \textit{lrev-work1} = \textit{lrev-work}

\langle proof \rangle

Now use the homomorphism.

\texttt{fixrec \textit{lrev-body2} :: ('a llist \to 'a H) \to 'a llist \to 'a H}

\texttt{where}

\texttt{\quad \textit{lrev-body2} \cdot r \cdot \textit{lnil} = ID}

\texttt{\quad | \textit{lrev-body2} \cdot r \cdot (x :@ xs) = \textit{list2H} \cdot \textit{wrapH} \cdot r \cdot xs \ \textit{oo} \ \textit{list2H} \cdot \textit{x :@ lnil}}

lemma \textit{lrev-body2-strict[simp]}: \textit{lrev-body2} \cdot \bot = \bot

\langle proof \rangle

definition \textit{lrev-work2} :: 'a llist \to 'a H

\texttt{where}

\texttt{\quad \textit{lrev-work2} \equiv \textit{fix} \cdot \textit{lrev-body2}}

\texttt{lemma \textit{lrev-work2-strict[simp]}: \textit{lrev-work2} \cdot \bot = \bot}

\langle proof \rangle

lemma \textit{lrev-work2-lrev-work1-eq}: \textit{lrev-work2} = \textit{lrev-work1}

\langle proof \rangle

Simplify.

\texttt{fixrec \textit{lrev-body3} :: ('a llist \to 'a H) \to 'a llist \to 'a H}

\texttt{where}

\texttt{\quad \textit{lrev-body3} \cdot r \cdot \textit{lnil} = ID}

\texttt{\quad | \textit{lrev-body3} \cdot r \cdot (x :@ xs) = r \cdot xs \ \textit{oo} \ \textit{list2H} \cdot \textit{x :@ lnil}}

lemma \textit{lrev-body3-strict[simp]}: \textit{lrev-body3} \cdot \bot = \bot

\langle proof \rangle

definition \textit{lrev-work3} :: 'a llist \to 'a H

\texttt{where}

\texttt{\quad \textit{lrev-work3} \equiv \textit{fix} \cdot \textit{lrev-body3}}

\texttt{lemma \textit{lrev-wwfusion}: \textit{list2H} \cdot \textit{((wrapH \cdot \textit{lrev-work2}) \cdot \textit{xs})} = \textit{lrev-work2} \cdot \textit{xs}}

\langle proof \rangle

If we use this result directly, we only get a partially-correct program transformation, see Tullsen (2002) for details.

\texttt{lemma \textit{lrev-work3} \subseteq \textit{lrev-work2}}

\langle proof \rangle

We can’t show the reverse inclusion in the same way as the fusion law doesn’t
hold for the optimised definition. (Intuitively we haven’t established that it is equal to the original \( \text{lrev} \) definition.) We could show termination of the optimised definition though, as it operates on finite lists. Alternatively we can use induction (over the list argument) to show total equivalence.

The following lemma shows that the fusion Gill/Hutton want to do is completely sound in this context, by appealing to the lazy list induction principle.

\[
\text{lemma } \text{lrev-work3-lrev-work2-eq: } \text{lrev-work3} = \text{lrev-work2} (\text{is } ?\text{lhs} = ?\text{rhs})
\]

⟨proof⟩

Use the combined worker/wrapper-fusion rule. Note we get a weaker lemma.

\[
\text{lemma } \text{lrev3-2-syntactic: } \text{lrev-body3 oo (unwrapH oo wrapH)} = \text{lrev-body2}
\]

⟨proof⟩

\[
\text{lemma } \text{lrev-work3-lrev-work2-eq: } \text{lrev} = \text{wrapH} \cdot \text{lrev-work3}
\]

⟨proof⟩

Final syntactic tidy-up.

\[
\text{fixrec } \text{lrev-body-final :: } ('a llist \rightarrow 'a \text{H}) \rightarrow 'a llist \rightarrow 'a \text{H where}
\]

\[
\begin{align*}
\text{lrev-body-final} \cdot \text{r} \cdot \text{lnil} \cdot \text{ys} & = \text{ys} \\
\text{lrev-body-final} \cdot \text{r} \cdot (x :@ \text{xs}) \cdot \text{ys} & = \text{r} \cdot \text{xs} (x :@ \text{ys})
\end{align*}
\]

\[
\text{definition } \text{lrev-work-final :: } 'a llist \rightarrow 'a \text{H where}
\]

\[
\text{lrev-work-final} \equiv \text{fix} \cdot \text{lrev-body-final}
\]

\[
\text{definition } \text{lrev-final :: } 'a llist \rightarrow 'a llist where
\]

\[
\text{lrev-final} \equiv \Lambda \text{ xs. lrev-work-final} \cdot \text{xs} \cdot \text{lnil}
\]

\[
\text{lemma } \text{lrev-body-final-lrev-body3-eq: } \text{lrev-body-final} \cdot \text{r} \cdot \text{xs} = \text{lrev-body3} \cdot \text{r} \cdot \text{xs}
\]

⟨proof⟩

\[
\text{lemma } \text{lrev-body-final-lrev-body3-eq: } \text{lrev-body-final} = \text{lrev-body3}
\]

⟨proof⟩

\[
\text{lemma } \text{lrev-final-lrev-eq: } \text{lrev} = \text{lrev-final} (\text{is } ?\text{lhs} = ?\text{rhs})
\]

⟨proof⟩

6 Unboxing types.

The original application of the worker/wrapper transformation was the unboxing of flat types by Peyton Jones and Launchbury (1991). We can model the boxed and unboxed types as (respectively) pointed and unpointed domains in HOLCF. Concretely \( \text{UNat} \) denotes the discrete domain of naturals,
\( UNat_\perp \) the lifted (flat and pointed) variant, and \( Nat \) the standard boxed domain, isomorphic to \( UNat_\perp \). This latter distinction helps us keep the boxed naturals and lifted function codomains separated; applications of unbox should be thought of in the same way as Haskell’s newtype constructors, i.e. operationally equivalent to \( ID \).

The divergence monad is used to handle the unboxing, see below.

6.1 Factorial example.

Standard definition of factorial.

fixrec fac :: Nat \to Nat
where
  fac \cdot n = If n =_B 0 then 1 else n \ast fac \cdot (n - 1)

declare fac.simps[simp del]

lemma fac-strict[simp]: fac \cdot \perp = \perp
(proof)

definition
  fac-body :: (Nat \to Nat) \to Nat \to Nat
  where
  fac-body \equiv \Lambda r n. If n =_B 0 then 1 else n \ast r \cdot (n - 1)

lemma fac-body-strict[simp]: fac-body \cdot r \cdot \perp = \perp
(proof)

lemma fac-fac-body-eq: fac = fix \cdot fac-body
(proof)

Wrap / unwrap functions. Note the explicit lifting of the co-domain. For some reason the published version of Gill and Hutton (2009) does not discuss this point: if we’re going to handle recursive functions, we need a bottom. unbox simply removes the tag, yielding a possibly-divergent unboxed value, the result of the function.

definition
  unwrapB :: (Nat \to Nat) \to UNat \to UNat_\perp
  where
  unwrapB \equiv \Lambda f. unbox oo f oo box

Note that the monadic bind operator (\( >>= \)) here stands in for the case construct in the paper.

definition
  wrapB :: (UNat \to UNat_\perp) \to Nat \to Nat
  where
  wrapB \equiv \Lambda f x. unbox \cdot x >>= f >>= box

lemma wrapB-unwrapB-body:
  assumes strictF: f \cdot \perp = \perp
shows \((\text{wrapB} \circ \text{unwrapB}) \circ f = f\) (is ?lhs = ?rhs)

(proof)

Apply worker/wrapper.

definition
\[ \text{fac-work} :: \text{UNat} \to \text{UNat} \_ \_ \_ \text{where} \]
\[ \text{fac-work} \equiv \text{fix} \langle \text{unwrapB} \circ \text{fac-body} \circ \text{wrapB} \rangle \]

definition
\[ \text{fac-wrap} :: \text{Nat} \to \text{Nat} \_ \_ \_ \text{where} \]
\[ \text{fac-wrap} \equiv \text{wrapB} \cdot \text{fac-work} \]

lemma\( \text{fac-fac-ww-eq} \): \(\text{fac} = \text{fac-wrap} \langle \text{is ?lhs} = \text{?rhs} \rangle\)

(proof)

This is not entirely faithful to the paper, as they don’t explicitly handle the lifting of the codomain.

definition
\[ \text{fac-body'} :: (\text{UNat} \to \text{UNat} \_ \_ \_ ) \to \text{UNat} \to \text{UNat} \_ \_ \_ \text{where} \]
\[ \text{fac-body'} \equiv \text{\Lambda} r \ n. \]
\[ \text{unbox} \cdot (\text{If} \ \text{box} \cdot n =_B 0 \]
\[ \text{then} \ 1 \]
\[ \text{else} \ \text{unbox} \cdot (\text{box} \cdot n - 1) \gg> r \gg> (\text{\Lambda} b. \ \text{box} \cdot n * \text{box} \cdot b)) \]

lemma\( \text{fac-body'} \cdot \text{fac-body}: \text{fac-body'} = \text{unwrapB} \circ \text{fac-body} \circ \text{wrapB} \langle \text{is ?lhs} = \text{?rhs} \rangle\)

(proof)

The \(\text{up}\) constructors here again mediate the isomorphism, operationally doing nothing. Note the switch to the machine-oriented \text{if} construct: the test \(n = (0 :: \text{a})\) cannot diverge.

definition
\[ \text{fac-body-final} :: (\text{UNat} \to \text{UNat} \_ \_ \_ ) \to \text{UNat} \to \text{UNat} \_ \_ \_ \text{where} \]
\[ \text{fac-body-final} \equiv \text{\Lambda} r \ n. \]
\[ \text{if} \ n = 0 \ \text{then} \ \text{up} \cdot 1 \ \text{else} \ r \cdot (n - \# 1) \gg> (\text{\Lambda} b. \ \text{up} \cdot (n * \# b)) \]

lemma\( \text{fac-body-final-fac-body'}: \text{fac-body-final} = \text{fac-body'} \langle \text{is ?lhs} = \text{?rhs} \rangle\)

(proof)

definition
\[ \text{fac-work-final} :: \text{UNat} \to \text{UNat} \_ \_ \_ \_ \text{where} \]
\[ \text{fac-work-final} \equiv \text{fix} \cdot \text{fac-body-final} \]

definition
\[ \text{fac-final} :: \text{Nat} \to \text{Nat} \_ \_ \_ \_ \_ \text{where} \]
\[ \text{fac-final} \equiv \text{\Lambda} n. \ \text{unbox} \cdot n \gg> \text{fac-work-final} \gg> \box \]

lemma\( \text{fac-fac-final}: \text{fac} = \text{fac-final} \langle \text{is ?lhs} = \text{?rhs} \rangle\)

(proof)
6.2 Introducing an accumulator.

The final version of factorial uses unboxed naturals but is not tail-recursive. We can apply worker/wrapper once more to introduce an accumulator, similar to §5.

The monadic machinery complicates things slightly here. We use Kleisli composition, denoted (>>=), in the homomorphism.

Firstly we introduce an “accumulator” monoid and show the homomorphism.

**type-synonym** UNatAcc = UNat → UNat⊥

**definition**

\[ n2a :: UNat \to UNatAcc \text{ where} \]
\[ n2a \equiv \Lambda m n. \ up\ (m \ast \# n) \]

**definition**

\[ a2n :: UNatAcc \to UNat⊥ \text{ where} \]
\[ a2n \equiv \Lambda a. a \cdot 1 \]

**lemma** a2n-strict[simp]: a2n⊥ = ⊥

(proof)

**lemma** a2n-n2a: a2n\( (n2a\cdot u) = up\cdot u \)

(proof)

**lemma** A-hom-mult: n2a\( (x \ast \# y) = (n2a\cdot x) >>= n2a\cdot y) \)

(proof)

**definition**

\[ unwrapA :: (UNat \to UNat⊥) \to UNat \to UNatAcc \text{ where} \]
\[ unwrapA \equiv \Lambda f n. f \cdot n >>= n2a \]

**lemma** unwrapA-strict[simp]: unwrapA⊥ = ⊥

(proof)

**definition**

\[ wrapA :: (UNat \to UNatAcc) \to UNat \to UNat⊥ \text{ where} \]
\[ wrapA \equiv \Lambda f. a2n \ oo f \]

**lemma** wrapA-unwrapA-id: wrapA oo unwrapA = ID

(proof)

Some steps along the way.

**definition**

\[ fac-acc-body1 :: (UNat \to UNatAcc) \to UNat \to UNatAcc \text{ where} \]
\[ fac-acc-body1 \equiv \Lambda r n. \]
\[ \text{if } n = 0 \text{ then } n2a \cdot 1 \text{ else } wrapA\cdot r\ (n - \# 1) >>= (\Lambda res. n2a\ (n \ast \# res)) \]
lemma fac-acc-body1-fac-body-final-eq: fac-acc-body1 = unwrapA oo fac-body-final oo wrapA
⟨proof ⟩

Use the homomorphism.

definition fac-acc-body2 :: (UNat → UNatAcc) → UNat → UNatAcc where
  fac-acc-body2 ≡ Λ r n.
  if n = 0 then n2a·1 else wrapA·r·(n − # 1) >>= (Λ res. n2a·n >>= n2a·res)

lemma fac-acc-body2-body1-eq: fac-acc-body2 = fac-acc-body1
⟨proof ⟩

Apply worker/wrapper.

definition fac-acc-body3 :: (UNat → UNatAcc) → UNat → UNatAcc where
  fac-acc-body3 ≡ Λ r n.
  if n = 0 then n2a·1 else n2a·n >>= r·(n − # 1)

lemma fac-acc-body3-body2: fac-acc-body3 oo (unwrapA oo wrapA) = fac-acc-body2
(is ?lhs=?rhs)
⟨proof ⟩

lemma fac-work-final-body3-eq: fac-work-final = wrapA·(fix·fac-acc-body3)
⟨proof ⟩

definition fac-acc-body-final :: (UNat → UNatAcc) → UNat → UNatAcc where
  fac-acc-body-final ≡ Λ r n acc.
  if n = 0 then up·acc else r·(n − # 1)·(n * # acc)

definition fac-acc-work-final :: UNat → UNat⊥ where
  fac-acc-work-final ≡ Λ x. fix·fac-acc-body-final·x·1

lemma fac-acc-work-final-fac-acc-work3-eq: fac-acc-body-final = fac-acc-body3
(is ?lhs=?rhs)
⟨proof ⟩

lemma fac-acc-work-final-fac-work: fac-acc-work-final = fac-work-final
(is ?lhs=?rhs)
⟨proof ⟩

7 Memoisation using streams.

7.1 Streams.

The type of infinite streams.
domain 'a Stream = stcons (lazy sthead :: 'a) (lazy sttail :: 'a Stream) (infixr & & 65)
⟨proof⟩

fixrec smap :: ('a → 'b) → 'a Stream → 'b Stream
where
smap.f·(x & & xs) = f·x & & smap.f·xs
⟨proof⟩

lemma smap-smap: smap.f·(smap.g·xs) = smap.(f oo g)·xs ⟨proof⟩

fixrec i-th :: 'a Stream → Nat → 'a
where
i-th·(x & & xs) = Nat-case·x·(i-th·xs)

abbreviation i-th-syn :: 'a Stream ⇒ Nat ⇒ 'a (infixl !! 100) where
s !! i ≡ i-th·s·i
⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩

The infinite stream of natural numbers.

fixrec nats :: Nat Stream
where
nats = 0 & & smap·(Λ x. 1 + x)·nats

7.2 The wrapper/unwrapper functions.

definition unwrapS' :: (Nat → 'a) → 'a Stream where
unwrapS' ≡ Λ f . smap.f·nats

lemma unwrapS'-unfold: unwrapS'·f = f·0 & & smap·(f oo (Λ x. 1 + x))·nats ⟨proof⟩

fixrec unwrapS :: (Nat → 'a) → 'a Stream
where
unwrapS·f = f·0 & & unwrapS·(f oo (Λ x. 1 + x))

The two versions of unwrapS are equivalent. We could try to fold some
definitions here but it’s easier if the stream constructor is manifest.

lemma unwrapS-unwrapS'-eq: unwrapS = unwrapS' (is ?lhs = ?rhs) ⟨proof⟩

definition wrapS :: 'a Stream → Nat → 'a where
wrapS ≡ Λ s i . s !! i

Note the identity requires that f be strict. Gill and Hutton (2009, §6.1) do
not make this requirement, an oversight on their part.

In practice all functions worth memoising are strict in the memoised argu-
ment.
lemma \( \text{wrapS-unwrapS-id}': \)
assumes strictF: \( f::\text{Nat} \to 'a \downarrow = \downarrow \)
shows unwrapS \( f \); n = f·n
⟨ proof ⟩

lemma \( \text{wrapS-unwrapS-id} : f \downarrow = \downarrow \implies (\text{wrapS oo unwrapS}) \cdot f = f \)
⟨ proof ⟩

7.3 Fibonacci example.

definition \( \text{fib-body} :: (\text{Nat} \to \text{Nat}) \to \text{Nat} \to \text{Nat} \ where \)
\( \text{fib-body} \equiv \Lambda r. \text{Nat-case} \cdot 1 \cdot (\text{Nat-case} \cdot 1 \cdot (\Lambda n. r \cdot n + r \cdot (n + 1))) \)
⟨ proof ⟩

definition \( \text{fib} :: \text{Nat} \to \text{Nat} \ where \)
\( \text{fib} \equiv \text{fix} \cdot \text{fib-body} \)
⟨ proof ⟩

Apply worker/wrapper.

definition \( \text{fib-work} :: \text{Nat Stream} \ where \)
\( \text{fib-work} \equiv \text{fix} \cdot (\text{unwrapS oo fib-body oo wrapS}) \)

definition \( \text{fib-wrap} :: \text{Nat} \to \text{Nat} \ where \)
\( \text{fib-wrap} \equiv \text{wrapS} \cdot \text{fib-work} \)

lemma \( \text{wrapS-unwrapS-fib-body} : \text{wrapS oo unwrapS} \circ \text{fib-body} = \text{fib-body} \)
⟨ proof ⟩

lemma \( \text{fib-ww-eq} : \text{fib} = \text{fib-wrap} \)
⟨ proof ⟩

Optimise.

fixrec \( \text{fib-work-final} :: \text{Nat Stream} \)
and \( \text{fib-f-final} :: \text{Nat} \to \text{Nat} \ where \)
\( \text{fib-work-final} = \text{smap} \cdot \text{fib-f-final-nats} \)
\( \text{fib-f-final} = \text{Nat-case} \cdot 1 \cdot (\text{Nat-case} \cdot 1 \cdot (\Lambda n'. \text{fib-work-final} ! n' + \text{fib-work-final} ! (n' + 1))) \)

declare \( \text{fib-f-final} .\text{simps}[\text{simp del}] \) \( \text{fib-work-final} .\text{simps}[\text{simp del}] \)

definition \( \text{fib-final} :: \text{Nat} \to \text{Nat} \ where \)
\( \text{fib-final} \equiv \Lambda n. \text{fib-work-final} ! n \)

18
This proof is only fiddly due to the way mutual recursion is encoded: we need to use Bekić’s Theorem (Bekić 1984) to massage the definitions into their final form.

**lemma fib-work-final-fib-work-eq**: fib-work-final = fib-work (is ?lhs = ?rhs)

**lemma fib-final-fib-eq**: fib-final = fib (is ?lhs = ?rhs)

8 Tagless interpreter via double-barreled continuations

type-synonym 'a Cont = ('a → 'a) → 'a

definition val2cont :: 'a → 'a Cont where
val2cont ≡ (Λ a c. c·a)

definition cont2val :: 'a Cont → 'a where
cont2val ≡ (Λ f. f·ID)

**lemma cont2val-val2cont-id**: cont2val oo val2cont = ID

domain Expr = Val (lazy val::Nat) | Add (lazy addl::Expr) (lazy addr::Expr) | Throw | Catch (lazy cbody::Expr) (lazy chandler::Expr)

**fixrec eval :: Expr → Nat Maybe**

where
  eval·(Val·n) = Just·n
  | eval·(Add·x·y) = mliftM2 (Λ a b. a + b)·(eval·x)·(eval·y)
  | eval·Throw = mfail
  | eval·(Catch·x·y) = mcatch·(eval·x)·(eval·y)

**fixrec eval-body :: (Expr → Nat Maybe) → Expr → Nat Maybe**

where
  eval-body·r·(Val·n) = Just·n
  | eval-body·r·(Add·x·y) = mliftM2 (Λ a b. a + b)·(r·x)·(r·y)
  | eval-body·r·Throw = mfail
  | eval-body·r·(Catch·x·y) = mcatch·(r·x)·(r·y)

1The interested reader can find some historical commentary in Harel (1980); Sangiorgi (2009).
lemma eval-body-strictExpr[simp]: eval-body·⊥ = ⊥
(proof)

lemma eval-eval-body-eq: eval = fix·eval-body
(proof)

8.1 Worker/wrapper
definition unwrapC :: (Expr → Nat Maybe) → (Expr → (Nat → Nat Maybe) → Nat Maybe)
where
unwrapC ≡ Λ g e s f. case g·e of Nothing ⇒ f | Just·n ⇒ s·n

lemma unwrapC-strict[simp]: unwrapC·⊥ = ⊥
(proof)
definition wrapC :: (Expr → (Nat → Nat Maybe) → Nat Maybe → Nat Maybe) → (Expr → Nat Maybe)
where
wrapC ≡ Λ g e. g·e·Just·Nothing

lemma wrapC-unwrapC-id: wrapC oo unwrapC = ID
(proof)
definition eval-work :: Expr → (Nat → Nat Maybe) → Nat Maybe → Nat Maybe where
eval-work ≡ fix·(unwrapC oo eval-body oo wrapC)
definition eval-wrap :: Expr → Nat Maybe where
eval-wrap ≡ wrapC·eval-work

fixrec eval-body’ :: (Expr → (Nat → Nat Maybe) → Nat Maybe → Nat Maybe)
where
eval-body’·r·(Val·n)·s·f = s·n
| eval-body’·r·(Add·x·y)·s·f = (case wrapC·r·x of
   Nothing ⇒ f
   | Just·n ⇒ (case wrapC·r·y of
     Nothing ⇒ f
     | Just·m ⇒ s·(n + m)))
| eval-body’·r·Throw·s·f = f
| eval-body’·r·(Catch·x·y)·s·f = (case wrapC·r·x of
   Nothing ⇒ (case wrapC·r·y of
     Nothing ⇒ f
     | Just·n ⇒ s·n)
   | Just·n ⇒ s·n)

lemma eval-body’-strictExpr[simp]: eval-body’·r·⊥·s·f = ⊥
This proof is unfortunately quite messy, due to the simplifier’s inability to cope with HOLCF’s case distinctions.

\[ \text{eval-body'} = \text{unwrapC} \circ \text{eval-body} \circ \text{wrapC} \]

\[ \text{eval-body-final} \equiv \text{fix} \circ \text{eval-body-final} \]

\[ \text{eval-final} \equiv (\Lambda e. \text{eval-work-final} \cdot e \cdot \text{Just} \cdot \text{Nothing}) \]

\[ \text{eval} = \text{eval-final} \]

9 Backtracking using lazy lists and continuations

To illustrate the utility of worker/wrapper fusion to programming language semantics, we consider here the first-order part of a higher-order backtracking language by Wand and Vaillancourt (2004); see also Danvy et al. (2001). We refer the reader to these papers for a broader motivation for these languages.

As syntax is typically considered to be inductively generated, with each syntactic object taken to be finite and completely defined, we define the syntax for our language using a HOL datatype:
The language consists of constants, an addition function, a disjunctive choice between expressions, and failure. We give it a direct semantics using the monad of lazy lists of natural numbers, with the goal of deriving an an extensionally-equivalent evaluator that uses double-barrelled continuations. Our theory of lazy lists is entirely standard.

default-sort predomain

domain 'a llist =
    lnil
  | lcons (lazy 'a) (lazy 'a llist)

By relaxing the default sort of type variables to predomain, our polymorphic definitions can be used at concrete types that do not contain ⊥. These include those constructed from HOL types using the discrete ordering type constructor 'a discr, and in particular our interpretation nat discr of the natural numbers.

The following standard list functions underpin the monadic infrastructure:

fixrec lappend :: 'a llist → 'a llist → 'a llist where
lappend·lnil·ys = ys
  | lappend·(lcons·x·xs)·ys = lcons·x·(lappend·xs·ys)

fixrec lconcat :: 'a llist llist → 'a llist where
lconcat·lnil = lnil
  | lconcat·(lcons·x·xs) = lappend·x·(lconcat·xs)

fixrec lmap :: ('a → 'b) → 'a llist → 'b llist where
lmap·f·lnil = lnil
  | lmap·f·(lcons·x·xs) = lcons·(f·x)·(lmap·f·xs)

We define the lazy list monad S in the traditional fashion:

type-synonym S = nat discr llist

definition returnS :: nat discr → S where
returnS = (Λ x. lcons·x·lnil)

definition bindS :: S → (nat discr → S) → S where
bindS = (Λ x g. lconcat·(lmap·g·x))

Unfortunately the lack of higher-order polymorphism in HOL prevents us from providing the general typing one would expect a monad to have in Haskell.

The evaluator uses the following extra constants:

definition addS :: S → S → S where
addS ≡ (Λ x y. bindS·x·(Λ xv. bindS·y·(Λ yv. returnS·(xv + yv))))
\textbf{definition} \textit{disjS} :: \( S \rightarrow S \rightarrow S \) where 
\( \text{disjS} \equiv \text{lappend} \)

\textbf{definition} \textit{failS} :: \( S \) where 
\( \text{failS} \equiv \text{lnil} \)

We interpret our language using these combinators in the obvious way. The only complication is that, even though our evaluator is primitive recursive, we must explicitly use the fixed point operator as the worker/wrapper technique requires us to talk about the body of the recursive definition.

\textbf{definition} \textit{evalS-body} :: \((\text{expr discr} \rightarrow \text{nat discr\ llist})\) \(\rightarrow\) \((\text{expr discr} \rightarrow \text{nat discr\ llist})\) where 
\( \text{evalS-body} \equiv \Lambda r\ e.\ \text{case undiscr\ e\ of} \) 
\( \text{const n} \Rightarrow \text{returnS} \cdot (\text{Discr n}) \) 
\( \text{add e1 e2} \Rightarrow \text{addS} \cdot (r \cdot (\text{Discr e1})) \cdot (r \cdot (\text{Discr e2})) \) 
\( \text{disj e1 e2} \Rightarrow \text{disjS} \cdot (r \cdot (\text{Discr e1})) \cdot (r \cdot (\text{Discr e2})) \) 
\( \text{fail} \Rightarrow \text{failS} \)

\textbf{abbreviation} \textit{evalS} :: \( \text{expr discr} \rightarrow \text{nat discr\ llist} \) where 
\( \text{evalS} \equiv \text{fix} \cdot \text{evalS-body} \)

We aim to transform this evaluator into one using double-barrelled continuations; one will serve as a "success" context, taking a natural number into "the rest of the computation", and the other outright failure.

In general we could work with an arbitrary observation type ala Reynolds (1974), but for convenience we use the clearly adequate concrete type \( \text{nat discr\ llist} \).

\textbf{type-synonym} \textit{Obs} = \( \text{nat discr\ llist} \)
\textbf{type-synonym} \textit{Failure} = \textit{Obs}
\textbf{type-synonym} \textit{Success} = \( \text{nat discr} \rightarrow \text{Failure} \rightarrow \text{Obs} \)
\textbf{type-synonym} \textit{K} = \( \text{Success} \rightarrow \text{Failure} \rightarrow \text{Obs} \)

To ease our development we adopt what Wand and Vaillancourt (2004, §5) call a "failure computation" instead of a failure continuation, which would have the type \( \text{unit} \rightarrow \text{Obs} \).

The monad over the continuation type \( K \) is as follows:

\textbf{definition} \textit{returnK} :: \( \text{nat discr} \rightarrow K \) where 
\( \text{returnK} \equiv (\Lambda x.\ \Lambda s\ f.\ s\cdot x\cdot f) \)

\textbf{definition} \textit{bindK} :: \( K \rightarrow (\text{nat discr} \rightarrow K) \rightarrow K \) where 
\( \text{bindK} \equiv (\Lambda x\ g.\ \Lambda s\ f.\ x\cdot (\Lambda x'\ f'.\ g\cdot x\cdot s\cdot f')\cdot f) \)

Our extra constants are defined as follows:

\textbf{definition} \textit{addK} :: \( K \rightarrow K \rightarrow K \) where
\(addK \equiv (\Lambda x \ y. \ bindK \cdot x \cdot (\Lambda xv. \ bindK \cdot y \cdot (\Lambda yv. \ returnK \cdot (xv + yv))))\)

**Definition**  
\(disjK :: K \rightarrow K \rightarrow K\) where  
\(disjK \equiv (\Lambda g \ h. \ \Lambda s f. \ g \cdot s \cdot (h \cdot s \cdot f))\)

**Definition**  
\(failK :: K\) where  
\(failK \equiv \Lambda s f. \ f\)

The continuation semantics is again straightforward:

**Definition**  
\(evalK-body :: (\text{expr discr} \rightarrow K) \rightarrow (\text{expr discr} \rightarrow K)\) where  
\(evalK-body \equiv \Lambda r \ e. \ \text{case undiscr e of}\)  
\(\mid \text{const n} \Rightarrow returnK \cdot (\text{Discr n})\)  
\(\mid \text{add e1 e2} \Rightarrow addK \cdot (r \cdot (\text{Discr e1})) \cdot (r \cdot (\text{Discr e2}))\)  
\(\mid \text{disj e1 e2} \Rightarrow disjK \cdot (r \cdot (\text{Discr e1})) \cdot (r \cdot (\text{Discr e2}))\)  
\(\mid \text{fail} \Rightarrow failK\)

**Abbreviation**  
\(evalK :: \text{expr discr} \rightarrow K\) where  
\(evalK \equiv \text{fix} \cdot evalK-body\)

We now set up a worker/wrapper relation between these two semantics.  
The kernel of \(unwrap\) is the following function that converts a lazy list into  
an equivalent continuation representation.

**Definition**  
\(fixrec \ SK :: S \rightarrow K\) where  
\(SK \cdot \text{lnil} = failK\)  
\(\mid SK \cdot \text{lcons} \cdot x \cdot xs = (\Lambda s f. \ s \cdot x \cdot (SK \cdot xs \cdot s \cdot f))\)

**Definition**  
\(unwrap :: (\text{expr discr} \rightarrow \text{nat discr llist}) \rightarrow (\text{expr discr} \rightarrow K)\) where  
\(unwrap \equiv \Lambda r \ e. \ SK \cdot (r \cdot e)\langle\text{proof}\rangle\langle\text{proof}\rangle\)

Symmetrically \(wrap\) converts an evaluator using continuations into one generating lazy lists by passing it the right continuations.

**Definition**  
\(KS :: K \rightarrow S\) where  
\(KS \equiv (\Lambda k. \ k \cdot \text{lcons} \cdot \text{lnil})\)

**Definition**  
\(wrap :: (\text{expr discr} \rightarrow K) \rightarrow (\text{expr discr} \rightarrow \text{nat discr llist})\) where  
\(wrap \equiv \Lambda r \ e. \ KS \cdot (r \cdot e)\langle\text{proof}\rangle\langle\text{proof}\rangle\)

The worker/wrapper condition follows directly from these definitions.

**Lemma**  
\(KS-SK-id:\)  
\(KS \cdot (SK \cdot xs) = xs\)  
\(\langle\text{proof}\rangle\)

**Lemma**  
\(wrap-unwrap-id:\)  
\(\text{wrap} \circ \text{unwrap} = \text{ID}\)

24
The worker/wrapper transformation is only non-trivial if \textit{wrap} and \textit{unwrap} do not witness an isomorphism. In this case we can show that we do not even have a Galois connection.

\begin{lemma}
\textit{cfun-not-below}:
\[
f \cdot x \nless g \cdot x \implies f \nless g
\]
\end{lemma}

\begin{lemma}
\textit{unwrap-wrap-not-under-id}:
\[
\text{unwrap oo wrap} \nless \text{ID}
\]
\end{lemma}

We now apply \textit{worker\_wrapper\_id}:

\begin{definition}
\textit{eval-work} :: \textit{expr discr} \rightarrow K\textit{ where}
\[
\text{eval-work} \equiv \text{fix}(\text{unwrap oo evalS-body oo wrap})
\]
\end{definition}

\begin{definition}
\textit{eval-ww} :: \textit{expr discr} \rightarrow \textit{nat discr list}\textit{ where}
\[
\text{eval-ww} \equiv \text{wrap} \cdot \text{eval-work}
\]
\end{definition}

\begin{lemma}
\textit{evalS = eval-ww}
\end{lemma}

We now show how the monadic operations correspond by showing that \textit{SK} witnesses a monad morphism (Wadler 1992, §6). As required by Danvy et al. (2001, Definition 2.1), the mapping needs to hold for our specific operations in addition to the common monadic scaffolding.

\begin{lemma}
\textit{SK-returnS-returnK}:
\[
\text{SK} \cdot (\text{returnS} \cdot x) = \text{returnK} \cdot x
\]
\end{lemma}

\begin{lemma}
\textit{SK-lappend-distrib}:
\[
\text{SK} \cdot (\text{lappend} \cdot xs \cdot ys) \cdot s \cdot f = \text{SK} \cdot xs \cdot s \cdot (\text{SK} \cdot ys \cdot s \cdot f)
\]
\end{lemma}

\begin{lemma}
\textit{SK-bindS-bindK}:
\[
\text{SK} \cdot (\text{bindS} \cdot x \cdot g) = \text{bindK} \cdot (\text{SK} \cdot x) \cdot (\text{SK} \circ g)
\]
\end{lemma}

\begin{lemma}
\textit{SK-addS-distrib}:
\[
\text{SK} \cdot (\text{addS} \cdot x \cdot y) = \text{addK} \cdot (\text{SK} \cdot x) \cdot (\text{SK} \cdot y)
\]
\end{lemma}

\begin{lemma}
\textit{SK-disjS-disjK}:
\[
\text{SK} \cdot (\text{disjS} \cdot xs \cdot ys) = \text{disjK} \cdot (\text{SK} \cdot xs) \cdot (\text{SK} \cdot ys)
\]
\end{lemma}

\begin{lemma}
\textit{SK-failS-failK}:
\[
\]
\]
These lemmas directly establish the precondition for our all-in-one worker/wrapper and fusion rule:

**Lemma evalS-body-evalK-body:**
\[ \text{unwrap oo evalS-body oo wrap} = \text{evalK-body oo unwrap oo wrap} \]

**Theorem evalS-evalK:**
\[ \text{evalS} = \text{wrap} \cdot \text{evalK} \]

This proof can be considered an instance of the approach of Hutton et al. (2010), which uses the worker/wrapper machinery to relate two algebras. This result could be obtained by a structural induction over the syntax of the language. However our goal here is to show how such a transformation can be achieved by purely equational means; this has the advantage that our proof can be locally extended, e.g. to the full language of Danvy et al. (2001) simply by proving extra equations. In contrast the higher-order language of Wand and Vaillancourt (2004) is beyond the reach of this approach.

## 10 Transforming $O(n^2)$ nub into an $O(n \lg n)$ one

Andy Gill’s solution, mechanised.

### 10.1 The nub function.

**Fixrec nub :: Nat list → Nat list**

**Where**
\[
\text{nub-nil} = \text{nil} \\
\text{nub}(x :@ \text{xs}) = x :@ \text{nub}((\text{filter} \cdot (\Lambda y. x = B y)) \cdot \text{xs})
\]

**Lemma nub-strict[simp]:** \(\text{nub-} \bot = \bot\)

**Fixrec nub-body :: (Nat list → Nat list) → Nat list → Nat list**

**Where**
\[
\text{nub-body} \cdot f \cdot \text{nil} = \text{nil} \\
\text{nub-body} \cdot f \cdot (x :@ \text{xs}) = x :@ f \cdot (\text{filter} \cdot (\Lambda y. x = B y)) \cdot \text{xs}
\]

**Lemma nub-nub-body-eq:** \(\text{nub} = \text{fix} \cdot \text{nub-body}\)
10.2 Optimised data type.

Implement sets using lazy lists for now. Lifting up HOL’s ‘a set type causes continuity grief.

**type-synonym** NatSet = Nat llist

**definition**

SetEmpty :: NatSet where
SetEmpty ≡ lnil

**definition**

SetInsert :: Nat → NatSet → NatSet where
SetInsert ≡ lcons

**definition**

SetMem :: Nat → NatSet → tr where
SetMem ≡ lmember · (bpred (=))

**lemma** SetMem-strict [simp]: SetMem · x · ⊥ = ⊥ ⟨proof⟩

**lemma** SetMem-SetEmpty [simp]: SetMem · x · SetEmpty = FF ⟨proof⟩

**lemma** SetMem-SetInsert: SetMem · v · (SetInsert · x · s) = (SetMem · v · s orelse x =B v) ⟨proof⟩

AndyG’s new type.

**domain** R = R (lazy resultR :: Nat llist) (lazy exceptR :: NatSet)

**definition**

nextR :: R → (Nat * R) Maybe where
nextR = (Λ r. case ldropWhile · (Λ x. SetMem · x · (exceptR · r)) · (resultR · r) of
  lnil ⇒ Nothing
  | x :@ xs ⇒ Just · (x, R · xs · (exceptR · r))))

**lemma** nextR-strict1 [simp]: nextR · ⊥ = ⊥ ⟨proof⟩

**lemma** nextR-strict2 [simp]: nextR · (R · ⊥ · s) = ⊥ ⟨proof⟩

**lemma** nextR-lnil[simp]: nextR · (R · lnil · s) = Nothing ⟨proof⟩

**definition**

filterR :: Nat → R → R where
filterR ≡ (Λ v r. R · (resultR · r) · (SetInsert · v · (exceptR · r)))

**definition**

c2a :: Nat llist → R where
c2a ≡ Λ xs. R · xs · SetEmpty

**definition**

a2c :: R → Nat llist where
\[ a2c \equiv \Lambda r. \text{lfilter}\left(\Lambda v. \text{neg}\left(\text{SetMem}\cdot v\cdot (\text{exceptR}\cdot r)\right)\right)\cdot \text{resultR}\cdot r \]

**Lemma:** \( a2c\text{-}\text{strict}\)[simp]: \( a2c \bot = \bot \) \( \langle \text{proof} \rangle \)

**Lemma:** \( a2c\text{-}c2a\text{-}id\): \( a2c \circ c2a = \text{ID} \) \( \langle \text{proof} \rangle \)

**Definition**
\[
\text{wrap} :: \left( \text{R} \rightarrow \text{Nat llist} \right) \rightarrow \text{Nat llist} \rightarrow \text{Nat llist} \ \text{where}
\text{wrap} \equiv \Lambda f \cdot (c2a \cdot f \cdot \text{resultR} \cdot r)
\]

**Definition**
\[
\text{unwrap} :: \left( \text{Nat llist} \rightarrow \text{Nat llist} \right) \rightarrow \text{R} \rightarrow \text{Nat llist} \ \text{where}
\text{unwrap} \equiv \Lambda f \cdot r \cdot (a2c \cdot f \cdot \text{resultR} \cdot r)
\]

**Lemma:** \( \text{unwrap}\text{-strict\text{-}simp}\): \( \text{unwrap} \cdot \bot = \bot \) \( \langle \text{proof} \rangle \)

**Lemma:** \( \text{wrap}\text{-unwrap\text{-}id}\): \( \text{wrap} \circ \text{unwrap} = \text{ID} \) \( \langle \text{proof} \rangle \)

Equivalences needed for later.

**Lemma:** \( \text{TR\text{-}deMorgan}\): \( \text{neg} \cdot (x \text{ orelse } y) = (\text{neg} \cdot x \text{ andalso neg} \cdot y) \) \( \langle \text{proof} \rangle \)

**Lemma**
\[
\text{case\text{-}maybe\text{-}case}: \ (\text{case} (\text{case } L \text{ of } \text{lnil } \Rightarrow \text{Nothing} \ | \ x :@ xs \Rightarrow \text{Just}(h \cdot x \cdot xs)) \text{ of} \ \text{Nothing } \Rightarrow f \ | \ \text{Just}(a, b) \Rightarrow g \cdot a \cdot b) = (\text{case } L \text{ of } \text{lnil } \Rightarrow f \ | \ x :@ xs \Rightarrow g \cdot (\text{fst}(h \cdot x \cdot xs)) \cdot (\text{snd}(h \cdot x \cdot xs)))
\]

**Lemma**
\[
\text{case\text{-}\text{a2c\text{-}case\text{-}caseR}}: \ (\text{case } a2c \cdot w \text{ of } \text{lnil } \Rightarrow f \ | \ x :@ xs \Rightarrow g \cdot x \cdot xs) = (\text{case } \text{nextR} \cdot w \text{ of } \text{Nothing } \Rightarrow f \ | \ \text{Just}(x, r) \Rightarrow g \cdot x \cdot (a2c \cdot r)) \ (\text{is } ?lhs = ?rhs) \ \langle \text{proof} \rangle
\]

**Lemma**
\[
\text{filter\text{-}filterR}: \text{lfilter}\left(\text{neg} \circ \left(\Lambda y. x =_{B} y\right)\right)\cdot (a2c \cdot r) = a2c\cdot \left(\text{filterR}\cdot x \cdot r\right)
\]

**Lemma**
\[\text{Apply worker/wrapper. Unlike Gill/Hutton, we manipulate the body of the worker into the right form then apply the lemma.}\]

**Definition**
\[
\text{nub\text{-}body} \equiv (R \rightarrow \text{Nat llist}) \rightarrow R \rightarrow \text{Nat llist} \ \text{where}
\text{nub\text{-}body} \equiv \Lambda f \cdot r \cdot \text{case } a2c \cdot r \text{ of } \text{lnil } \Rightarrow \text{lnil} \ | \ x :@ xs \Rightarrow x :@ f \cdot (c2a \cdot (\text{lfilter}\cdot (\text{neg} \circ \left(\Lambda y. x =_{B} y\right))) \cdot xs))
\]

**Lemma**
\[
\text{nub\text{-}body\text{-nub\text{-}body}\text{-eq}}: \text{unwrap} \circ \text{nub\text{-}body} \circ \text{unwrap} = \text{nub\text{-}body}\'
\]


definition
nub-body'' :: (R → Nat llist) → R → Nat llist where
nub-body'' ≡ Λ f r. case nextR r of Nothing ⇒ lnil
| Just (x, xs) ⇒ x :@ f · (c2a · (lfilter (neg oo (Λ y z. x =B y)) · (a2c · xs))))

lemma nub-body''-nub-body'''-eq: nub-body'' = nub-body'''
⟨proof⟩

definition
nub-body'''' :: (R → Nat llist) → R → Nat llist where
nub-body'''' ≡ (Λ f r. case nextR r of Nothing ⇒ lnil
| Just (x, xs) ⇒ x :@ f · (filterR x · xs))

lemma nub-body''''-nub-body''''-eq: nub-body'''' = nub-body'''' oo (unwrap oo wrap)
⟨proof⟩

Finally glue it all together.
lemma nub-wrap-nub-body''''': nub = wrap · (fix · nub-body''''')
⟨proof⟩

end

11 Optimise “last”.

Andy Gill’s solution, mechanised. No fusion, works fine using their rule.

11.1 The last function.

fixrec llast :: 'a llist → 'a
where
llast (x :@ yys) = (case yys of lnil ⇒ x | y :@ yys ⇒ llast · yys)

lemma llast-strict[simp]: llast ⊥ = ⊥
⟨proof⟩

fixrec llast-body :: ('a llist → 'a) → 'a llist → 'a
where
llast-body f (x :@ yys) = (case yys of lnil ⇒ x | y :@ yys ⇒ f · yys)

lemma llast-llast-body: llast = fix · llast-body
⟨proof⟩

definition wrap :: ('a → 'a llist → 'a) → ('a llist → 'a) where
wrap ≡ Λ f (x :@ xs). f · x · xs
**definition** `unwrap :: ('a list → 'a) → ('a → 'a list → 'a) where`

`unwrap ≡ Λ f x xs. f·(x :@ xs)`

**lemma** `unwrap-strict [simp]: unwrap⊥ = ⊥`

⟨proof⟩

**lemma** `wrap-unwrap-ID: wrap oo unwrap oo llast-body = llast-body`

⟨proof⟩

**definition** `llast-worker :: ('a → 'a list → 'a) → 'a → 'a llist → 'a where`

`llast-worker ≡ Λ r x yys. case yys of lnil ⇒ x | y :@ ys ⇒ r·y·ys`

**definition** `llast' :: 'a list → 'a where`

`llast' ≡ wrap·(fix·llast-worker)`

**lemma** `llast-worker-llast-body: llast-worker = unwrap oo llast-body oo wrap`

⟨proof⟩

**lemma** `llast'·llast: llast' = llast (is ?lhs = ?rhs)`

⟨proof⟩

end

## 12 Concluding remarks

Gill and Hutton provide two examples of fusion: accumulator introduction in their §4, and the transformation in their §7 of an interpreter for a language with exceptions into one employing continuations. Both involve strict `unwrap`s and are indeed totally correct.

The example in their §5 demonstrates the unboxing of numerical computations using a different worker/wrapper rule and does not require fusion. In their §6 a non-strict `unwrap` is used to memoise functions over the natural numbers using the rule considered here. It should in fact use the same rule as the unboxing example as the scheme only correctly memoises strict functions. We can see this by considering a base case missing from their inductive proof, viz that if `f :: Nat → a` is not strict – in fact constant, as `Nat` is a flat domain – then `f ⊥ ≠ ⊥ = (map f [0..]) !! ⊥`, where `xs !! n` is the `n`th element of `xs`.

### References


