Mechanising the worker/wrapper transformation

Peter Gammie

September 19, 2015

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

10.1 The nub function. ................................................. 26
10.2 Optimised data type. ............................................. 26

11 Optimise "last".

11.1 The last function. ................................................. 29

12 Concluding remarks ............................................... 30

Bibliography .......................................................... 30

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. ⟨ML⟩

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: $\text{fix}(g \circ f) \subseteq g \cdot (\text{fix}(f \circ g))$
(proof)

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

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

\[
\begin{array}{c}
\bullet \\
\downarrow g \\
\bullet \\
\bullet \\
\downarrow f \\
\bullet \\
\end{array}
\quad \xrightarrow{\text{fix}}
\begin{array}{c}
\bullet \\
\downarrow g \\
\bullet \\
\bullet \\
\downarrow \text{fix } h \\
\bullet \\
\end{array}
\]

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 \texttt{parallel-fix-ind}:  

\[
\text{adm} \left( \lambda x. \ ?P \left( \text{fst } x \right) \left( \text{snd } x \right) \right) \\
?P \perp \perp \\
\bigwedge x y. \ ?P x y \\
\xrightarrow{?P \left( \text{fix} \cdot \text{?}F \right) \left( \text{fix} \cdot \text{?}G \right)}
\]

in a very straightforward way:

\textbf{lemma lfp-fusion:}
\begin{itemize}
  \item assumes \( g \perp = \perp \)
  \item assumes \( g \circ \circ f = h \circ \circ g \)
  \item shows \( g \circ (\text{fix} \cdot f) = \text{fix} \cdot h \)
\end{itemize}

\textlangle proof \textrangle

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

\textlangle proof \textrangle \textlangle proof \textrangle \textlangle proof \textrangle \textlangle proof \textrangle \textlangle proof \textrangle \textlangle proof \textrangle

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:

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

(the worker/wrapper transformation)

Also:

\[
(\text{unwrap} \circ \text{wrap}) \cdot \text{work} = \text{work}
\]

(worker/wrapper fusion)

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

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

**Lemma** \( \text{worker/wrapper-id} \):

- \( \text{fixes} \) \( \text{wrap} :: B \to A \)
- \( \text{fixes} \) \( \text{unwrap} :: A \to B \)
- \( \text{assumes} \) \( \text{wrap} \circ \text{unwrap} = \text{id} \)
- \( \text{assumes} \) \( \text{comp-body: computation} = \text{fix} \cdot \text{body} \)
- \( \text{shows} \) \( \text{computation} = \text{wrap} \cdot \left( \text{fix} \circ \text{unwrap} \circ \text{comp} \circ \text{body} \circ \text{wrap} \right) \)

\( \langle \text{proof} \rangle \)

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

**Lemma** \( \text{worker/wrapper-body} \):

- \( \text{fixes} \) \( \text{wrap} :: B \to A \)
- \( \text{fixes} \) \( \text{unwrap} :: A \to B \)
- \( \text{assumes} \) \( \text{wrap} \circ \text{unwrap} \circ \text{body} = \text{body} \)
- \( \text{assumes} \) \( \text{comp-body: computation} = \text{fix} \cdot \text{body} \)
- \( \text{shows} \) \( \text{computation} = \text{wrap} \cdot \left( \text{fix} \circ \text{unwrap} \circ \text{body} \right) \)

\( \langle \text{proof} \rangle \)

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** \( \text{worker/wrapper-fix} \):

- \( \text{fixes} \) \( \text{wrap} :: B \to A \)
- \( \text{fixes} \) \( \text{unwrap} :: A \to B \)
- \( \text{assumes} \) \( \text{fix} \circ \text{unwrap} \circ \text{body} = \text{fix} \cdot \text{body} \)
- \( \text{assumes} \) \( \text{comp-body: computation} = \text{fix} \cdot \text{body} \)
- \( \text{shows} \) \( \text{computation} = \text{wrap} \cdot \left( \text{fix} \circ \text{unwrap} \circ \text{body} \right) \)
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** \textit{worker-wrapper-fusion}:
- \textit{fixes} \texttt{wrap} :: \texttt{'}b::pcpo \to \texttt{'}a::pcpo
- \textit{fixes} \texttt{unwrap} :: \texttt{'}a \to \texttt{'}b
- \textit{assumes} \texttt{wrap-unwrap}: \texttt{wrap} \circ \texttt{unwrap} = \texttt{ID}
- \textit{assumes} \texttt{work}: \texttt{work} = \texttt{fix}(\texttt{unwrap} \circ \texttt{body} \circ \texttt{wrap})
- \textit{shows} \((\text{unwrap} \circ \text{wrap}) \cdot \text{work} = \text{work}

\(<\text{proof}>\)

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 \texttt{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 \texttt{fold} rule treats this as a set of rewrite rules oriented right-to-left. (The \texttt{unfold} rule treats the current working set of definitions as rewrite rules oriented left-to-right.) Hence as each definition \(f = \texttt{body}\) yields a rule of the form \(\texttt{body} \implies f\), 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 \texttt{fold} rule is Park induction:

\[
\frac{f \cdot ?x = ?x}{\text{fix.f} \subseteq ?x} \text{ fix least}
\]

viz that \(f \cdot x = x\) 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 \texttt{unwrap} is non-strict (i.e. \(\texttt{unwrap} \perp \neq \perp\)) 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 \( \text{body}' \) such that the body of the worker, i.e. the expression \( \text{unwrap} \circ \text{body} \circ \text{wrap} \), can be rewritten as \( \text{body}' \circ \text{unwrap} \circ \text{wrap} \). Intuitively this is the semantic form of workers where all self-calls are fusible. Our goal is to justify redefining \( \text{work} \) to \( \text{fix} \cdot \text{body}' \), i.e. to establish:

\[
\text{fix}(\text{unwrap} \circ \text{body} \circ \text{wrap}) = \text{fix} \cdot \text{body}'
\]

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

\[
\text{lemma fusion-partially-correct:}
\begin{align*}
\text{assumes wrap-unwrap:} & \quad \text{wrap} \circ \text{unwrap} = \text{ID} \\
\text{assumes work:} & \quad \text{work} = \text{fix}(\text{unwrap} \circ \text{body} \circ \text{wrap}) \\
\text{assumes body':} & \quad \text{unwrap} \circ \text{body} \circ \text{wrap} = \text{body}' \circ \text{unwrap} \circ \text{wrap} \\
\text{shows} & \quad \text{fix} \cdot \text{body}' \subseteq \text{work}
\end{align*}
\]

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 \( A \).

\[
\text{domain} \quad A = A \\
\text{domain} \quad B = B (\text{lazy} \ A)
\]

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 \( \forall x. \ ?f \cdot (\ ?g \cdot x) = x \implies ?f \cdot \bot = \bot \).

\[
\text{fixrec} \quad \text{wrap} :: B \rightarrow A \\
\text{where} \quad \text{wrap} \cdot (B \cdot a) = a \\
\langle \text{proof} \rangle
\]

\[
\text{fixrec} \quad \text{unwrap} :: A \rightarrow B \\
\text{where} \quad \text{unwrap} = B
\]

Discharging the worker/wrapper hypothesis is similarly routine.

\[
\text{lemma wrap-unwrap:} \quad \text{wrap} \circ \text{unwrap} = \text{ID} \\
\langle \text{proof} \rangle
\]

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

\[
\text{fixrec} \quad \text{body} :: A \rightarrow A \\
\text{where} \quad \text{body} \cdot r = A
\]
The wrinkle is that the transformed worker can be strict in the recursion parameter $r$, as $unwrap$ always lifts it.

```haskell
fixrec body' :: B -> B
where body' · (r · a) = B · A(proof)
```

As explained above, we set up the fusion opportunity:

```haskell
lemma body-body': unwrap oo body oo wrap = body' oo unwrap oo wrap
(proof)
```

This result depends crucially on $unwrap$ being non-strict.

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

```haskell
lemma fix-body' subset fix · (unwrap oo body oo wrap)
(proof)
```

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

```haskell
lemma ~ fix · (unwrap oo body oo wrap) subset fix-body'
(proof)
```

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

Note that if we have already established that $wrap oo unwrap = ID$, then making $unwrap$ strict preserves this equation:

```haskell
lemma
  assumes wrap oo unwrap = ID
  shows wrap oo strictify · unwrap = ID
(proof)
```

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

(proof)

### 4 A totally-correct fusion rule

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

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

```haskell
lemma worker$-wrapper$-fusion-new:
```
For a recursive definition \( \text{comp} = \text{body} \) of type \( A \) and a pair of functions \( \text{wrap} :: B \to A \) and \( \text{unwrap} :: A \to B \) where \( \text{wrap} \circ \text{unwrap} = \text{id}_A \) and \( \text{unwrap} \bot = \bot \), define:

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

(the worker/wrapper transformation)

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

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

(worker/wrapper fusion)

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

\begin{align*}
\text{fixes} \quad \text{wrap} :: 'b::\text{pcpo} & \to 'a::\text{pcpo} \\
\text{fixes} \quad \text{unwrap} :: 'a & \to 'b \\
\text{fixes} \quad \text{body} :: 'b & \to 'b \\
\text{assumes} \quad \text{wrap-unwrap}: \text{wrap} \circ \text{unwrap} &= (\text{id} :: 'a \to 'a) \\
\text{assumes} \quad \text{unwrap-strict}: \text{unwrap} \bot = \bot \\
\text{assumes} \quad \text{body-body}': \text{unwrap oo body oo wrap} &= \text{body'} oo (\text{unwrap oo wrap}) \\
\text{shows} \quad \text{fix-body} &= (\text{wrap oo body'})
\end{align*}

\( \langle \text{proof} \rangle \)

We can also show a more general result which allows fusion to be optionally performed on a per-recursive-call basis using \texttt{parallel_fix_ind}:

\begin{align*}
\text{lemma} \quad \text{worker-wrapper-fusion-new-general}: \\
\text{fixes} \quad \text{wrap} :: 'b::\text{pcpo} & \to 'a::\text{pcpo} \\
\text{fixes} \quad \text{unwrap} :: 'a & \to 'b \\
\text{assumes} \quad \text{wrap-unwrap}: \text{wrap} \circ \text{unwrap} &= (\text{id} :: 'a \to 'a) \\
\text{assumes} \quad \text{unwrap-strict}: \text{unwrap} \bot = \bot \\
\text{assumes} \quad \text{body-body}': \forall r. (\text{unwrap oo body oo wrap}) \cdot r = r \\
\text{shows} \quad \text{fix-body} &= (\text{wrap oo body'})
\end{align*}

\( \langle \text{proof} \rangle \)

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 \textit{lrev-body} as the generator of the fixpoint definition of \textit{lrev} directly.

\begin{proof}

\end{proof}

\begin{lemma}
\textit{lrev-lrev-body-eq}: \textit{lrev} = \textit{fix-lrev-body}
\end{lemma}

Wrap / unwrap functions.

\begin{definition}
\textit{unwrapH} :: ('a list \to 'a list) \to 'a list \to 'a H where \\
\textit{unwrapH} \equiv \Lambda f xs . \textit{list2H} \cdot (f \cdot xs)
\end{definition}

\begin{lemma}
\textit{unwrapH-strict[simp]}: \textit{unwrapH} \cdot \bot = \bot
\end{lemma}

\begin{definition}
\textit{wrapH} :: ('a list \to 'a H) \to 'a list \to 'a list where \\
\textit{wrapH} \equiv \Lambda f xs . \textit{H2list} \cdot (f \cdot xs)
\end{definition}

\begin{lemma}
\textit{wrapH-unwrapH-id}: \textit{wrapH} oo \textit{unwrapH} = \textit{ID} (is \textit{?lhs} = \textit{?rhs})
\end{lemma}

\section{Gill/Hutton-style worker/wrapper.}

\begin{definition}
\textit{lrev-work} :: 'a list \to 'a H where \\
\textit{lrev-work} \equiv \textit{fix} \cdot (\textit{unwrapH} oo \textit{lrev-body} oo \textit{wrapH})
\end{definition}

\begin{definition}
\textit{lrev-wrap} :: 'a list \to 'a list where \\
\textit{lrev-wrap} \equiv \textit{wrapH} \cdot \textit{lrev-work}
\end{definition}

\begin{lemma}
\textit{lrev-lrev-ww-eq}: \textit{lrev} = \textit{lrev-wrap}
\end{lemma}

\section{Optimise worker/wrapper.}

Intermediate worker.

\begin{fixrec}
\textit{lrev-body1} :: ('a list \to 'a H) \to 'a list \to 'a H where \\
\textit{lrev-body1} \cdot r \cdot \textit{lnil} = \textit{list2H} \cdot \textit{lnil} \\
| \textit{lrev-body1} \cdot r \cdot (x :@ xs) = \textit{list2H} \cdot (\textit{wrapH} \cdot r \cdot xs :++ (x :@ \textit{lnil}))
\end{fixrec}

\begin{definition}
\textit{lrev-work1} :: 'a list \to 'a H where \\
\textit{lrev-work1} \equiv \textit{fix-lrev-body1}
\end{definition}

\begin{lemma}
\textit{lrev-body-lrev-body1-eq}: \textit{lrev-body1} = \textit{unwrapH} oo \textit{lrev-body} oo \textit{wrapH}
\end{lemma}
lemma lrev-work1-lrev-work-eq: lrev-work1 = lrev-work
⟨proof⟩

Now use the homomorphism.

fixrec lrev-body2 :: (′a llist → ′a H) → ′a llist → ′a H
where
  lrev-body2·r·nil = ID
| lrev-body2·r·(x :@ xs) = list2H·(wrapH·r·xs) oo list2H·(x :@ nil)

lemma lrev-body2-strict[simp]: lrev-body2·⊥ = ⊥
⟨proof⟩

definition lrev-work2 :: ′a llist → ′a H where
  lrev-work2 ≡ fix·lrev-body2

lemma lrev-work2-strict[simp]: lrev-work2·⊥ = ⊥
⟨proof⟩

lemma lrev-body2-lrev-body1-eq: lrev-body2 = lrev-body1
⟨proof⟩

lemma lrev-work2-lrev-work1-eq: lrev-work2 = lrev-work1
⟨proof⟩

Simplify.

fixrec lrev-body3 :: (′a llist → ′a H) → ′a llist → ′a H
where
  lrev-body3·r·nil = ID
| lrev-body3·r·(x :@ xs) = r·xs oo list2H·(x :@ nil)

lemma lrev-body3-strict[simp]: lrev-body3·⊥ = ⊥
⟨proof⟩

definition lrev-work3 :: ′a llist → ′a H where
  lrev-work3 ≡ fix·lrev-body3

lemma lrev-wwfusion: list2H·((wrapH·lrev-work2)·xs) = lrev-work2·xs
⟨proof⟩

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

lemma lrev-work3 ⊑ lrev-work2
⟨proof⟩

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 \textit{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.

\textbf{lemma} \textit{lrev-work3-lrev-work2-eq}: \textit{lrev-work3} = \textit{lrev-work2} (\texttt{is \ ?lhs = \ ?rhs})

\textbf{⟨proof⟩}

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

\textbf{lemma} \textit{lrev3-2-syntactic}: \textit{lrev-body3 oo (unwrapH oo wrapH)} = \textit{lrev-body2}

\textbf{⟨proof⟩}

\textbf{lemma} \textit{lrev-work3-lrev-work2-eq‘}: \textit{lrev} = \textit{wrapH \cdot lrev-work3}

\textbf{⟨proof⟩}

Final syntactic tidy-up.

\textbf{fixrec} \textit{lrev-body-final :: (\texttt{\'a llist} \rightarrow \texttt{\'a H}) \rightarrow \texttt{\'a llist} \rightarrow \texttt{\'a H}}

\textbf{where}

\textit{lrev-body-final \cdot \texttt{r \cdot lnil} \cdot ys} = \texttt{ys}

\textit{| lrev-body-final \cdot (x \vdash \texttt{xs}) \cdot ys} = \texttt{r \cdot (x \vdash \texttt{ys})}

\textbf{definition}

\textit{lrev-work-final :: \texttt{\'a llist} \rightarrow \texttt{\'a H}}

\textit{lrev-work-final ≡ fix \cdot lrev-body-final}

\textbf{definition}

\textit{lrev-final :: \texttt{\'a llist} \rightarrow \texttt{\'a llist}}

\textit{lrev-final ≡ λ \texttt{xs. lrev-work-final \cdot xs \cdot lnil}}

\textbf{lemma} \textit{lrev-body-final-lrev-body3-eq‘}: \textit{lrev-body-final \cdot r \cdot xs} = \textit{lrev-body3 \cdot r \cdot xs}

\textbf{⟨proof⟩}

\textbf{lemma} \textit{lrev-body-final-lrev-body3-eq}: \textit{lrev-body-final} = \textit{lrev-body3}

\textbf{⟨proof⟩}

\textbf{lemma} \textit{lrev-final-lrev-eq}: \textit{lrev} = \textit{lrev-final} (\texttt{is \ ?lhs = \ ?rhs})

\textbf{⟨proof⟩}

\section{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 \texttt{UNat} denotes the discrete domain of naturals,
UNat⊥ the lifted (flat and pointed) variant, and Nat the standard boxed
domain, isomorphic to UNat⊥. This latter distinction helps us keep the
boxed naturals and lifted function codomains separated; applications of un-
box 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.

\[\text{fixrec fac :: Nat} \to \text{Nat} \]
\[\text{where} \quad \text{fac-} n = \text{If } n = \text{B } 0 \text{ then } 1 \text{ else } n \ast \text{fac-}(n - 1)\]

declare fac.simps[simp del]

\[\text{lemma fac-strict}[\text{simp}]: \text{fac-} \bot = \bot\]
\[\langle \text{proof} \rangle\]

definition \[\text{fac-body} :: (Nat} \to \text{Nat}) \to \text{Nat} \to \text{Nat} \text{ where} \]
\[\text{fac-body} \equiv \Lambda r \ n. \text{If } n = \text{B } 0 \text{ then } 1 \text{ else } n \ast r \cdot (n - 1)\]

\[\text{lemma fac-body-strict}[\text{simp}]: \text{fac-body-r} \cdot \bot = \bot\]
\[\langle \text{proof} \rangle\]

\[\text{lemma fac-fac-body-eq}: \text{fac} = \text{fix} \cdot \text{fac-body}\]
\[\langle \text{proof} \rangle\]

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 \[\text{unwrapB} :: (Nat} \to \text{Nat}) \to \text{UNat} \to \text{UNat} \bot \text{ where} \]
\[\text{unwrapB} \equiv \Lambda f \ . \ \text{unbox oo f oo box}\]

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

definition \[\text{wrapB} :: (\text{UNat} \to \text{UNat} \bot) \to \text{Nat} \to \text{Nat} \text{ where} \]
\[\text{wrapB} \equiv \Lambda f \ . \ \text{unbox} \cdot x \gg= f \gg= \text{box}\]

\[\text{lemma wrapB-unwrapB-body}: \text{assumes strictF} : f \cdot \bot = \bot\]
shows (wrapB oo unwrapB)\cdot f = f (is ?lhs = ?rhs)
(proof)

Apply worker/wrapper.

definition
fac-work :: UNat \to UNat_\bot where
fac-work \equiv \text{fix (unwrapB oo fac-body oo wrapB)}

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

lemma fac-fac-ww-eq: fac = fac-wrap (is ?lhs = ?rhs)
(proof)

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

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

lemma fac-body'-fac-body: fac-body' = unwrapB oo fac-body oo wrapB (is ?lhs = ?rhs)
(proof)

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

definition
fac-body-final :: (UNat \to UNat_\bot) \to UNat \to UNat_\bot where
fac-body-final \equiv \Lambda r n.
  if n = 0 then up\cdot 1 else r\cdot(n - # 1) \gg (\Lambda b. up\cdot (n \cdot# b))

lemma fac-body-final-fac-body': fac-body-final = fac-body' (is ?lhs = ?rhs)
(proof)

definition
fac-work-final :: UNat \to UNat_\bot where
fac-work-final \equiv \text{fix \cdot fac-body-final}

definition
fac-final :: Nat \to Nat where
fac-final \equiv \Lambda n. unbox\cdot n \gg fac-work-final \gg = box

lemma fac-fac-final: fac = fac-final (is ?lhs = ?rhs)
(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 $op \gg\gg$, in the homomorphism.

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

**type-synonym** $UNatAcc = UNat \to UNat_\bot$

**definition**

$n2a :: UNat \to UNatAcc$ where

$n2a \equiv \Lambda m n. \ up\ (m *\# n)$

**definition**

$a2n :: UNatAcc \to UNat_\bot$ where

$a2n \equiv \Lambda a. a \cdot 1$

**lemma** $a2n\text{-strict}[simp]$: $a2n\cdot \bot = \bot$

(proof)

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

(proof)

**lemma** $A\text{-hom-mult}$: $n2a\cdot (x *\# y) = (n2a\cdot x \gg\gg n2a\cdot y)$

(proof)

**definition**

$unwrapA :: (UNat \to UNat_\bot) \to UNat \to UNatAcc$ where

$unwrapA \equiv \Lambda f n. f \cdot n \gg\gg n2a$

**lemma** $unwrapA\text{-strict}[simp]$: $unwrapA\cdot \bot = \bot$

(proof)

**definition**

$wrapA :: (UNat \to UNatAcc) \to UNat \to UNat_\bot$ where

$wrapA \equiv \Lambda f. a2n \ oo f$

**lemma** $wrapA\cdot unwrapA\text{-id}$: $wrapA \ oo \ unwrapA = ID$

(proof)

Some steps along the way.

**definition**

$fac\text{-acc\text{-}body1} :: (UNat \to UNatAcc) \to UNat \to UNatAcc$ where

$fac\text{-acc\text{-}body1} \equiv \Lambda r n. $

$\text{if } n = 0 \text{ then } n2a\cdot 1 \text{ else } wrapA\cdot r\cdot (n - \# 1) \gg\gg (\Lambda res. n2a\cdot (n *\# res))$
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 \rightarrow UNatAcc) \rightarrow UNat \rightarrow 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 \rightarrow UNatAcc) \rightarrow UNat \rightarrow 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 \rightarrow UNatAcc) \rightarrow UNat \rightarrow 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 \rightarrow 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 wrapS-unwrapS-id':
  assumes strictF: (f::Nat → 'a)·⊥ = ⊥
  shows unwrapS·f !! n = f·n
⟨proof⟩

lemma wrapS-unwrapS-id: f·⊥ = ⊥ =⇒ (wrapS oo unwrapS)·f = f
⟨proof⟩

7.3 Fibonacci example.

definition fib-body :: (Nat → Nat) → Nat → Nat where
  fib-body ≡ Λ r. Nat-case·1·(Nat-case·1·(Λ n· r·n + r·(n + 1)))
⟨proof⟩

definition fib :: Nat → Nat where
  fib ≡ fix·fib-body
⟨proof⟩

Apply worker/wrapper.

definition fib-work :: Nat Stream where
  fib-work ≡ fix·(unwrapS oo fib-body oo wrapS)

definition fib-wrap :: Nat → Nat where
  fib-wrap ≡ wrapS·fib-work

lemma wrapS-unwrapS-fib-body: wrapS oo unwrapS oo fib-body = fib-body
⟨proof⟩

lemma fib-ww-eq: fib = fib-wrap
⟨proof⟩

Optimise.

fixrec
  fib-work-final :: Nat Stream
and
  fib-f-final :: Nat → Nat
where
  fib-work-final = smap·fib-f-final-nats
| fib-f-final = Nat-case·1·(Nat-case·1·(Λ n′. fib-work-final !! n′ + fib-work-final !! (n′ + 1)))
declare fib-f-final.simps[simp del] fib-work-final.simps[simp del]

definition fib-final :: Nat → Nat where
  fib-final ≡ Λ n. 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)\(^1\) to massage the definitions into their final form.

**lemma** fib-work-final-fib-work-eq: fib-work-final = fib-work (is ?lhs = ?rhs)  
(\langle proof}\)

**lemma** fib-final-fib-eq: fib-final = fib (is ?lhs = ?rhs)  
(\langle proof}\

### 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  
(\langle proof}\

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

fixrec eval :: Expr → Nat Maybe  
where  
\[
eval·(\Val·n) = \Just·n \\
\text{eval·(Add·x·y)} = \text{mliftM2} (\Lambda a\ b. a + b)·(eval·x)·(eval·y) \\
\text{eval·Throw} = \text{mfail} \\
\text{eval·(Catch·x·y)} = \text{mcatch}·(eval·x)·(eval·y)
\]

fixrec eval-body :: (Expr → Nat Maybe) → Expr → Nat Maybe  
where  
\[
eval-body·r·(\Val·n) = \Just·n \\
\text{eval-body·r·(Add·x·y)} = \text{mliftM2} (\Lambda a\ b. a + b)·(r·x)·(r·y) \\
\text{eval-body·r·Throw} = \text{mfail} \\
\text{eval-body·r·(Catch·x·y)} = \text{mcatch}·(r·x)·(r·y)
\]

\(^1\)The 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) → (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.

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

\[ \text{eval-body-final} = \text{unwrapC oo eval-body oo wrapC} \]

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

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 =

| nnil |
| 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·fnil = lnil
| lmap·f·(lcons·x·xs) = lcons·(f·x)·(lmap·f·xs)\(\langle\text{proof}\rangle\langle\text{proof}\rangle\langle\text{proof}\rangle)

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

**type-synonym** $S = \text{nat discr llist}$

**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 
\textit{disjS} \equiv \text{lappend}

\textbf{definition} \textit{failS} :: \(S\) where 
\textit{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 
\textit{evalS-body} \equiv \Lambda \text{e} \cdot \begin{cases} 
\text{returnS} \cdot (\text{Discr e}) & | \text{const n} \\
\text{addS} \cdot (\text{r} \cdot (\text{Discr e1})) \cdot (\text{r} \cdot (\text{Discr e2})) & | \text{add e1 e2} \\
\text{disjS} \cdot (\text{r} \cdot (\text{Discr e1})) \cdot (\text{r} \cdot (\text{Discr e2})) & | \text{disj e1 e2} \\
\text{failS} & | \text{fail} 
\end{cases}

\textbf{abbreviation} \textit{evalS} :: \text{expr discr} \rightarrow \text{nat discr llist} where 
\textit{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} = \text{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 \text{K} is as follows:

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

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

Our extra constants are defined as follows:

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

**definition** \textit{disjK} :: \( K \rightarrow K \rightarrow K \) where
\[
\text{disjK} \equiv (\Lambda g h. \Lambda s f. g\cdot s\cdot (h\cdot s\cdot f))
\]

**definition** \textit{failK} :: \( K \) where
\[
\text{failK} \equiv \Lambda s f. f
\]

The continuation semantics is again straightforward:

**definition** \textit{evalK-body} :: \((\text{expr discr} \rightarrow K) \rightarrow (\text{expr discr} \rightarrow K)\) where
\[
\text{evalK-body} \equiv \Lambda r e. \text{case undiscr e of}
\begin{align*}
&\text{const n} \Rightarrow \text{returnK}\cdot (\text{Discr n}) \\
&\text{add e1 e2} \Rightarrow \text{addK}\cdot (r\cdot (\text{Discr e1}))\cdot (r\cdot (\text{Discr e2})) \\
&\text{disj e1 e2} \Rightarrow \text{disjK}\cdot (r\cdot (\text{Discr e1}))\cdot (r\cdot (\text{Discr e2})) \\
&\text{fail} \Rightarrow \text{failK}
\end{align*}
\]

**abbreviation** \textit{evalK} :: \( \text{expr discr} \rightarrow K \) where
\[
\text{evalK} \equiv \text{fix}\cdot \text{evalK-body}
\]

We now set up a worker/wrapper relation between these two semantics.

The kernel of \textit{unwrap} is the following function that converts a lazy list into an equivalent continuation representation.

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

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

Symmetrically \textit{wrap} converts an evaluator using continuations into one generating lazy lists by passing it the right continuations.

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

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

The worker/wrapper condition follows directly from these definitions.

**lemma** \textit{KS-SK-id}:
\[
\text{KS}\cdot (\text{SK}\cdot xs) = xs \langle \text{proof} \rangle
\]

**lemma** \textit{wrap-unwrap-id}:
\[
\text{wrap oo unwrap} = \text{ID}
\]
The worker/wrapper transformation is only non-trivial if \texttt{wrap} and \texttt{unwrap} do not witness an isomorphism. In this case we can show that we do not even have a Galois connection.

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

\begin{lemma}
\textbf{unwrap-wrap-not-under-id:}
\[ \texttt{unwrap} \circ \texttt{wrap} \not\sqsubseteq \texttt{ID} \]
\end{lemma}

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

\begin{definition}
\texttt{eval-work} :: expr discr \to K \textbf{where}
\texttt{eval-work} \equiv \mathit{fix}\,(\texttt{unwrap} \circ \texttt{evalS\_body} \circ \texttt{wrap})
\end{definition}

\begin{definition}
\texttt{eval-ww} :: expr discr \to \texttt{nat discr list} \textbf{where}
\texttt{eval-ww} \equiv \texttt{wrap} \cdot \texttt{eval-work}
\end{definition}

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

We now show how the monadic operations correspond by showing that \texttt{SK} witnesses a \textit{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}
\textbf{SK-returnS-returnK:}
\[ \texttt{SK} \cdot (\texttt{returnS} \cdot x) = \texttt{returnK} \cdot x \]
\end{lemma}

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

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

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

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

\begin{lemma}
\textbf{SK-failS-failK:}
\[ \texttt{SK} \cdot \texttt{failS} = \texttt{failK} \]
\end{lemma}
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:**
\[ \text{evalS} = \text{wrap-evalK} \]

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 \(O(n^2)\) \textit{nub} into an \(O(n \lg n)\) one

Andy Gill’s solution, mechanised.

### 10.1 The \textit{nub} function.

**Fixrec** \textit{nub} :: \textit{Nat list} \to \textit{Nat list}

**Where**
\begin{align*}
\text{nub-nil} = \text{nil} \\
| \text{nub}(x :@ \text{xs}) = x :@ \text{nub}(\text{filter}(\text{neg oo } (\Lambda y. x \equiv_B y)) \cdot \text{xs})
\end{align*}

**Lemma** \textit{nub-strict[simp]}: \textit{nub-}\top = \top

**Fixrec** \textit{nub-body} :: (\textit{Nat list} \to \textit{Nat list}) \to \textit{Nat list} \to \textit{Nat list}

**Where**
\begin{align*}
\text{nub-body-f-nil} = \text{nil} \\
| \text{nub-body-f}(x :@ \text{xs}) = x :@ f(\text{filter}(\text{neg oo } (\Lambda y. x \equiv_B y)) \cdot \text{xs})
\end{align*}

**Lemma** \textit{nub-nub-body-eq}:
\[ \text{nub} = \text{fix-nub-body} \]
10.2 Optimised data type.

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

**type-synonym** \( \text{NatSet} = \text{Nat llist} \)

**definition**
\[
\text{SetEmpty} :: \text{NatSet} \text{ where} \\
\text{SetEmpty} \equiv \text{lnil}
\]

**definition**
\[
\text{SetInsert} :: \text{Nat} \to \text{NatSet} \to \text{NatSet} \text{ where} \\
\text{SetInsert} \equiv \text{lcons}
\]

**definition**
\[
\text{SetMem} :: \text{Nat} \to \text{NatSet} \to \text{tr} \text{ where} \\
\text{SetMem} \equiv \text{bmember} \cdot ( \text{bpred} \ (\text{op} =))
\]

**lemma** \( \text{SetMem}\text{-strict}[\text{simp}] : \text{SetMem}\cdot x \cdot \bot = \bot \) \( (\text{proof}) \)

**lemma** \( \text{SetMem}\text{-SetEmpty}[\text{simp}] : \text{SetMem}\cdot x \cdot \text{SetEmpty} = \text{FF} \) \( (\text{proof}) \)

**lemma** \( \text{SetMem}\text{-SetInsert} : \text{SetMem}\cdot v \cdot (\text{SetInsert}\cdot x \cdot s) = (\text{SetMem}\cdot v \cdot s \text{ orelse } x =_B v) \) \( (\text{proof}) \)

AndyG’s new type.

**domain** \( R = R \text{ (lazy} \text{ resultR} :: \text{Nat llist}) \text{ (lazy} \text{ exceptR} :: \text{NatSet}) \)

**definition**
\[
\text{nextR} :: R \to (\text{Nat} \ast R) \text{ Maybe} \text{ where} \\
\text{nextR} = (\Lambda r. \text{case ldropWhile} \cdot (\Lambda x. \text{SetMem}\cdot x \cdot (\text{exceptR}\cdot r))\cdot (\text{resultR}\cdot r) \text{ of} \\
\text{lnil} \Rightarrow \text{Nothing} \\
| x :\oplus xs \Rightarrow \text{Just} \cdot (x, R\cdot xs \cdot (\text{exceptR}\cdot r)))
\]

**lemma** \( \text{nextR}\text{-strict1}[\text{simp}] : \text{nextR}\cdot \bot = \bot \) \( (\text{proof}) \)

**lemma** \( \text{nextR}\text{-strict2}[\text{simp}] : \text{nextR}\cdot (R\cdot \bot \cdot S) = \bot \) \( (\text{proof}) \)

**lemma** \( \text{nextR}\text{-Init}[\text{simp}] : \text{nextR}\cdot (R\cdot \text{Init}\cdot S) = \text{Nothing} \) \( (\text{proof}) \)

**definition**
\[
\text{filterR} :: \text{Nat} \to R \to R \text{ where} \\
\text{filterR} \equiv (\Lambda v r. R\cdot \text{resultR}\cdot r \cdot (\text{SetInsert}\cdot v \cdot (\text{exceptR}\cdot r)))
\]

**definition**
\[
\text{c2a} :: \text{Nat llist} \to R \text{ where} \\
\text{c2a} \equiv \text{lnil} \ast \text{SetEmpty}
\]

**definition**
\[
\text{a2c} :: R \to \text{Nat llist} \text{ where}
\]
\[ a2c \equiv \Lambda \ r. \ \text{filter} \cdot (\Lambda \ v. \ \text{neg} \cdot (\text{SetMem} \cdot v \cdot (\text{exceptR} \cdot r))) \cdot (\text{resultR} \cdot r) \]

**lemma** \(a2c\)-strict[simp]: \(a2c \perp = \perp\)  
(proof)

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

**definition**
\[
\begin{align*}
\text{wrap} & : (R \to \text{Nat llist}) \to \text{Nat llist} \to \text{Nat llist} \text{ where} \\
\text{wrap} & \equiv \Lambda f \ \text{xs} \cdot f \cdot (\text{c2a} \cdot \text{xs})
\end{align*}
\]

**definition**
\[
\begin{align*}
\text{unwrap} & : (\text{Nat llist} \to \text{Nat llist}) \to R \to \text{Nat llist} \text{ where} \\
\text{unwrap} & \equiv \Lambda f \ r. \ f \cdot (a2c \cdot r)
\end{align*}
\]

**lemma** \(\text{unwrap}\)-strict[simp]: \(\text{unwrap} \cdot \perp = \perp\)  
(proof)

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

Equivalences needed for later.

**lemma** \(\text{TR-deMorgan}\): \(\text{neg} \cdot (x \lor y) = (\text{neg} \cdot x \land \text{neg} \cdot y)\)  
(proof)

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

**lemma** case-a2c-case-caseR:
\[
\begin{align*}
\text{(case a2c \cdot w of lnil \Rightarrow f} \mid x :\@ \text{xs} \Rightarrow g \cdot x \cdot \text{xs})
\end{align*}
\]
\[
\begin{align*}
= \text{(case nextR \cdot w of Nothing \Rightarrow f} \mid \text{Just} \cdot (x, r) \Rightarrow g \cdot x \cdot (a2c \cdot r)) \text{ (is ?lhs = ?rhs)}
\end{align*}
\]
(proof)

**lemma** filter-filterR: \(\text{filter} \cdot (\text{neg oo} \ (\Lambda y. \ x = B \ y)) \cdot (a2c \cdot r) = a2c \cdot (\text{filterR} \cdot x \cdot r)\)  
(proof)

Apply worker/wrapper. Unlike Gill/Hutton, we manipulate the body of the worker into the right form then apply the lemma.

**definition**
\[
\begin{align*}
\text{nub-body} \equiv (R \to \text{Nat llist}) \to R \to \text{Nat llist} \text{ where} \\
\text{nub-body} \equiv \Lambda f \ r. \ \text{case a2c \cdot r of lnil \Rightarrow lnil} \\
\mid x :\@ \text{xs} \Rightarrow x :\@ f \cdot (\text{c2a} \cdot (\text{filter} \cdot (\text{neg oo} \ (\Lambda y. \ x = B \ y)) \cdot \text{xs}))
\end{align*}
\]

**lemma** nub-body-nub-body'-eq: \(\text{unwrap oo nub-body oo wrap} = \text{nub-body}'\)
\begin{proof}

definition \text{nub-body''} :: (\mathbb{R} \to \text{Nat list}) \to \mathbb{R} \to \text{Nat list} where
\text{nub-body''} \equiv \Lambda f \cdot \text{case nextR-r of Nothing } \Rightarrow \text{lnil}
\quad | \text{Just}(x, xs) \Rightarrow x :\mathbb{R} \cdot f \cdot (c2a \cdot (\text{filter} \cdot (\text{neg oo} (\Lambda y \cdot x =_{\mathbb{R}} y))) :: (a2c \cdot xs)))

lemma \text{nub-body''-nub-body''-eq}: \text{nub-body''} = \text{nub-body''}
\langle \text{proof} \rangle

definition \text{nub-body'''} :: (\mathbb{R} \to \text{Nat list}) \to \mathbb{R} \to \text{Nat list} where
\text{nub-body'''} \equiv (\Lambda f \cdot \text{case nextR-r of Nothing } \Rightarrow \text{lnil}
\quad | \text{Just}(x, xs) \Rightarrow x :\mathbb{R} \cdot f \cdot (\text{filterR} \cdot x \cdot xs))

lemma \text{nub-body''-nub-body''-eq}: \text{nub-body'''} = \text{nub-body''} oo (\text{unwrap oo wrap})
\langle \text{proof} \rangle

Finally glue it all together.

lemma \text{nub-wrap-nub-body''': nub = wrap \cdot (fix \cdot \text{nub-body'''})}
\langle \text{proof} \rangle
\end{proof}

\section{11 Optimise \text{“last”}.}

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

\subsection{11.1 The \text{last} function.}

\begin{proof}

fixrec \text{llast} :: 'a llist \to 'a
where
\text{llast}(x :\mathbb{R} \cdot yys) = (\text{case yys of lnil } \Rightarrow x \mid y :\mathbb{R} \cdot ys \Rightarrow \text{llast}-yys)

lemma \text{llast-strict[simp]}: \text{llast} \cdot \bot = \bot
\langle \text{proof} \rangle

fixrec \text{llast-body} :: ('a llist \to 'a) \to 'a llist \to 'a
where
\text{llast-body.f}(x :\mathbb{R} \cdot yys) = (\text{case yys of lnil } \Rightarrow x \mid y :\mathbb{R} \cdot ys \Rightarrow f \cdot yys)

lemma \text{llast-llast-body}: \text{llast} = \text{fix \cdot llast-body}
\langle \text{proof} \rangle

\end{proof}
**definition** unwrap :: \(\forall \text{a llist } \rightarrow \text{a} \rightarrow (\text{'a llist } \rightarrow \text{a})\) where
\[
\text{unwrap } \equiv \Lambda f \ x \ xs. \ f(x :@ xs)
\]

**lemma** unwrap-strict[simp]: unwrap \(\bot = \bot\)

\(\langle\text{proof}\rangle\)

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

\(\langle\text{proof}\rangle\)

**definition** llast-worker :: \(\forall \text{a llist } \rightarrow \text{a} \rightarrow \text{a llist } \rightarrow \text{a}\) where
\[
\text{llast-worker } \equiv \Lambda r \ x \ yys. \ \text{case} \ yys \ \text{of lnil } \Rightarrow x \mid y \ :@ ys \Rightarrow r \cdot y \cdot ys
\]

**definition** llast' :: \(\forall \text{a llist } \rightarrow \text{a}\) where
\[
\text{llast'} \equiv \text{wrap} \cdot \text{fix-llast-worker}
\]

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

\(\langle\text{proof}\rangle\)

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

\(\langle\text{proof}\rangle\)

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 unwraps 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 \(f :: \text{Nat} \rightarrow \text{a}\) is not strict – in fact constant, as \(\text{Nat}\) is a flat domain – then \(f \bot \neq \bot = (\text{map} \ f \ [0..]) \ !! \bot\), where \(xs \ !! n\) is the \(n\)th element of \(xs\).

References


31


