Mechanising the worker/wrapper transformation

Peter Gammie

August 28, 2014

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 4
  3.1 Worker/wrapper fusion is partially correct .......... 6
  3.2 A non-strict \texttt{unwrap} may go awry .............. 7

4 A totally-correct fusion rule 9

5 Naive reverse becomes accumulator-reverse. 11
  5.1 Hughes lists, naive reverse, worker-wrapper optimisation. 11
  5.2 Gill/Hutton-style worker/wrapper. .......................... 12
  5.3 Optimise worker/wrapper. .................................. 13

6 Unboxing types. 17
  6.1 Factorial example. ..................................... 17
  6.2 Introducing an accumulator. ............................. 20

7 Memoisation using streams. 22
  7.1 Streams. .............................................. 22
  7.2 The wrapper/unwrapper functions. ...................... 23
  7.3 Fibonacci example. .................................... 24

8 Tagless interpreter via double-barreled continuations 26
  8.1 Worker/wrapper ................................. 27

9 Backtracking using lazy lists and continuations 29
10 Transforming $O(n^2)$ \textit{nub} into an $O(n \lg n)$ one

10.1 The \textit{nub} function. ........................................... 34
10.2 Optimised data type. .............................................. 35

11 Optimise “last”.

11.1 The \textit{last} function. ........................................... 38

12 Concluding remarks .................................................. 39

Bibliography ............................................................. 40

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 \textit{oo} stands for function composition, \textit{Λ} for continuous function abstraction, \textit{·} for continuous function application, \textit{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 \textit{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).

\textbf{lemma} rolling-rule-ltr: fix\cdot(g oo f) ⊑ g\cdot(fix\cdot(f oo g))

\textbf{proof} –

\begin{itemize}
  \item have g\cdot(fix\cdot(f oo g)) ⊑ g\cdot(fix\cdot(f oo g))
    \begin{itemize}
      \item by (rule below-refl) — reflexivity
    \end{itemize}
  \item hence g\cdot((f oo g)\cdot(fix\cdot(f oo g))) ⊑ g\cdot(fix\cdot(f oo g))
    \begin{itemize}
      \item using \textit{fix-eq[where F=f oo g]} by \textit{simp} — computation
    \end{itemize}
  \item hence (g oo f)\cdot(g\cdot(fix\cdot(f oo g))) ⊑ g\cdot(fix\cdot(f oo g))
    \begin{itemize}
      \item by \textit{simp} — re-associate \textit{op oo}
    \end{itemize}
  \item thus fix\cdot(g oo f) ⊑ g\cdot(fix\cdot(f oo g))
\end{itemize}
using \texttt{fix-least-below} by \texttt{blast} — induction

\texttt{qed}

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

\texttt{proof} —

\texttt{have} \(\text{fix} \cdot (f \circ \circ g) \sqsubseteq f \cdot (\text{fix} \cdot (g \circ \circ f))\) \texttt{by} \texttt{(rule} rolling-rule-ltr\texttt{)}

\texttt{hence} \(g \cdot (\text{fix} \cdot (f \circ \circ g)) \sqsubseteq g \cdot (f \cdot (\text{fix} \cdot (g \circ \circ f)))\)

\texttt{by} \texttt{(rule} monofun-cfun-arg\texttt{)} — \(g\) is monotonic

\texttt{thus} \(g \cdot (\text{fix} \cdot (f \circ \circ g)) \sqsubseteq \text{fix} \cdot (g \circ \circ f)\)

\texttt{using} \texttt{fix-eq[where} \(F = g \circ \circ f)\] \texttt{by} \texttt{simp} — computation

\texttt{qed}

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

\texttt{by} \texttt{(rule} below-antisym[\texttt{OF} rolling-rule-ltr rolling-rule-rtl\texttt{]}\)

\subsection{2.2 Least-fixed-point fusion}

\textit{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{center}
\begin{tikzpicture}
  \node (h) {\(h\)};
  \node (g) [below of=h] {\(g\)};
  \node (f) [left of=g] {\(f\)};
  \node (fix-h) [above right of=h] {\text{fix} \(h\)};
  \node (fix-f) [below right of=f] {\text{fix} \(f\)};
  \draw[->] (h) to (g);
  \draw[->] (g) to (f);
  \draw[->] (f) to (g);
  \draw[->] (g) to (h);
  \node at (0.5,0) {\(\Rightarrow\)};
\end{tikzpicture}
\end{center}

Fokkinga and Meijer (1991), and also their later Meijer, Fokkinga, and Patterson (1991), made extensive use of this rule, as did Tullsen (2002) in his program transformation tool \textsc{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}:

\[
\begin{array}{c}
\text{adm} (\lambda x. \ ?P \ (\text{fst} \ x) \ (\text{snd} \ x)) \\
\ ?P \perp \perp \quad \bigwedge_{x \ y} \ ?P \ x \ y \\
\ ?P \ (\text{fix} \cdot \ ?F) \ (\text{fix} \cdot \ ?G)
\end{array}
\]

in a very straightforward way:

\texttt{lemma} \texttt{lfp-fusion}:

\begin{itemize}
  \item \texttt{assumes} \(g \perp = \perp\)
  \item \texttt{assumes} \(g \circ \circ f = h \circ \circ g\)
  \item \texttt{shows} \(g \cdot (\text{fix} \cdot f) = \text{fix} \cdot h\)
\end{itemize}

\texttt{proof} (\texttt{induct rule:} parallel-fix-ind)
For a recursive definition \( \text{comp} = \text{fix body} \) 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:

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

(the worker/wrapper transformation)

Also:

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

(worker/wrapper fusion)

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

\begin{verbatim}
case 2 show g·⊥ = ⊥ by fact
case (3 x y)
  from g·x = y; g oo f = h oo g; show g·(f·x) = h·y
  by (simp add: cfun-eq-iff)
qed simp
\end{verbatim}

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

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.

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

\begin{verbatim}
lemma worker-wrapper-id:
  fixes \( \text{wrap} :: 'b::pcpo \to 'a::pcpo \)
  fixes \( \text{unwrap} :: 'a \to 'b \)
  assumes \( \text{wrap-unwrap}: \text{wrap} oo \text{unwrap} = \text{ID} \)
  assumes \( \text{comp-body}: \text{computation} = \text{fix-body} \)
  shows \( \text{computation} = \text{wrap-}(\text{fix-}(\text{unwrap oo body oo wrap})) \)
proof –
  from \( \text{comp-body} \) have \( \text{computation} = \text{fix-}(\text{ID oo body}) \)
\end{verbatim}
by simp
also from wrap-unwrap have ... = fix·(wrap oo unwrap oo body)
   by (simp add: assoc-oo)
also have ... = wrap·(fix·(unwrap oo body oo wrap))
   using rolling-rule[where f=unwrap oo body and g=wrap]
   by (simp add: assoc-oo)
finally show ?thesis .
qed

The second weakens this assumption by requiring that wrap oo wrap only act as the identity on values in the image of body.

lemma worker-wrapper-body:
  fixes wrap :: 'b::pcpo → 'a::pcpo
  fixes unwrap :: 'a → 'b
  assumes wrap-unwrap: wrap oo unwrap oo body = body
  assumes comp-body: computation = fix·body
  shows computation = wrap·(fix·(unwrap oo body oo wrap))
proof –
  from comp-body have computation = fix·(wrap oo unwrap oo body)
     using wrap-unwrap by (simp add: assoc-oo wrap-unwrap)
  also have ... = wrap·(fix·(unwrap oo body oo wrap))
     using rolling-rule[where f=unwrap oo body and g=wrap]
     by (simp add: assoc-oo)
  finally show ?thesis .
qed

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 wrap :: 'b::pcpo → 'a::pcpo
  fixes unwrap :: 'a → 'b
  assumes wrap-unwrap: fix·(wrap oo unwrap oo body) = fix·body
  assumes comp-body: computation = fix·body
  shows computation = wrap·(fix·(unwrap oo body oo wrap))
proof –
  from comp-body have computation = fix·(wrap oo unwrap oo body)
     using wrap-unwrap by (simp add: assoc-oo wrap-unwrap)
  also have ... = wrap·(fix·(unwrap oo body oo wrap))
     using rolling-rule[where f=unwrap oo body and g=wrap]
     by (simp add: assoc-oo)
  finally show ?thesis .
qed

Gill and Hutton’s worker-wrapper-fusion rule is intended to allow the transformation of (unwrap oo wrap)·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** \(\text{wrap} :: 'b::pcpo \rightarrow 'a::pcpo\)
- **fixes** \(\text{unwrap} :: 'a \rightarrow 'b\)
- **assumes** \(\text{wrap-unwrap}: \text{wrap} \circ \text{unwrap} = \text{ID}\)
- **assumes** \(\text{work}: \text{work} = \text{fix} \circ (\text{unwrap} \circ \text{body} \circ \text{wrap})\)
- **shows** \((\text{unwrap} \circ \text{wrap}) \circ \text{work} = \text{work}\)

**proof**

1. **have** \((\text{unwrap} \circ \text{wrap}) \circ \text{work} = (\text{unwrap} \circ \text{wrap}) \circ \text{fix} \circ (\text{unwrap} \circ \text{body} \circ \text{wrap})\)
   - **using** \(\text{work by simp}\)
   - **also have** \((\text{unwrap} \circ \text{wrap}) \circ \text{fix} \circ (\text{unwrap} \circ \text{body} \circ \text{wrap} \circ \text{unwrap} \circ \text{wrap})\)
     - **using** \(\text{wrap-unwrap by (simp add: assoc-oo)}\)
     - **also have** \((\text{unwrap} \circ \text{wrap} \circ \text{unwrap} \circ \text{body} \circ \text{wrap})\)
       - **using** \(\text{rolling-rule[where f=unwrap oo body oo wrap and g=unwrap oo wrap]}\)
         - **by** \((\text{simp add: assoc-oo)}\)
     - **also have** \((\text{unwrap} \circ \text{body} \circ \text{wrap})\)
       - **using** \(\text{wrap-unwrap by (simp add: assoc-oo)}\)
   - **finally show** \(\text{thesis using work by simp}\)

**qed**

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 \(\text{f = body}\) yields a rule of the form \(\text{body} \Rightarrow \text{f}\), one can always derive \(\text{f = f}\). 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:

\[
\begin{align*}
\text{f} \cdot \forall x = \forall x \\
\text{fix} \cdot f \sqsubseteq \forall x
\end{align*}
\]
viz that \( f \cdot x = x \) implies only the partially correct \( \text{fix } f \sqsubseteq x \), and not the totally correct \( \text{fix } f = x \). We use this characterisation to show that if \( \text{unwrap} \) is non-strict (i.e. \( \text{unwrap} \perp \neq \perp \)) 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:

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

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

**lemma fusion-partially-correct:**

- **assumes** \( \text{wrap-unwrap} \): \( \text{wrap } \circ \text{unwrap} = \text{ID} \)
- **assumes** \( \text{work} \): \( \text{work} = \text{fix } \cdot (\text{unwrap } \circ \text{body} \circ \text{wrap}) \)
- **assumes** \( \text{body'} \): \( \text{unwrap } \circ \text{body} \circ \text{wrap} = \text{body'} \circ \text{unwrap } \circ \text{wrap} \)
- **shows** \( \text{fix } \cdot \text{body'} \sqsubseteq \text{work} \)

**proof** (rule fix-least)

- **have** \( \text{work} = (\text{unwrap } \circ \text{body} \circ \text{wrap}) \cdot \text{work} \)
  - **using** \( \text{work} \) by (simp add: fix-eq[symmetric])
  - **also have** \( ... = (\text{body'} \circ \text{unwrap } \circ \text{wrap}) \cdot \text{work} \)
    - **using** \( \text{body'} \) by simp
    - **also have** \( ... = (\text{body'} \circ \text{unwrap } \circ \text{wrap}) \cdot ((\text{unwrap } \circ \text{body} \circ \text{wrap}) \cdot \text{work}) \)
      - **using** \( \text{work} \) by (simp add: fix-eq[symmetric])
      - **also have** \( ... = (\text{body'} \circ \text{unwrap } \circ \text{wrap} \circ \text{unwrap } \circ \text{body} \circ \text{wrap}) \cdot \text{work} \)
        - **by** simp
    - **also have** \( ... = (\text{body'} \circ \text{unwrap } \circ \text{body} \circ \text{wrap}) \cdot \text{work} \)
      - **using** \( \text{wrap-unwrap} \) by (simp add: assoc-oo)
      - **also have** \( ... = \text{body'} \cdot \text{work} \)
        - **using** \( \text{work} \) by (simp add: fix-eq[symmetric])
  - **finally show** \( \text{body'} \cdot \text{work} = \text{work} \) by simp

**qed**

The next section shows the converse does not obtain.

### 3.2 A non-strict \( \text{unwrap} \) may go awry

If \( \text{unwrap} \) 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 \( A \).

**domain** \( A = A \)
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 \(\forall x. \; \text{\texttt{?f}} \cdot (\text{\texttt{?g}} \cdot x) = x \implies \text{\texttt{?f}} \cdot \bot = \bot\).

\begin{verbatim}
fixrec wrap :: B \to A
where wrap \cdot (B \cdot a) = a

fixrec unwrap :: A \to B
where unwrap = B
\end{verbatim}

Discharging the worker/wrapper hypothesis is similarly routine.

\begin{verbatim}
lemma wrap-unwrap: wrap oo unwrap = ID
  by (simp add: cfun-eq-iff)
\end{verbatim}

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

\begin{verbatim}
fixrec body :: A \to A
where body \cdot r = A
\end{verbatim}

The wrinkle is that the transformed worker can be strict in the recursion parameter \(r\), as \textit{unwrap} always lifts it.

\begin{verbatim}
fixrec body' :: B \to B
where body' \cdot (B \cdot a) = B \cdot A
\end{verbatim}

As explained above, we set up the fusion opportunity:

\begin{verbatim}
lemma body-body': unwrap oo body oo wrap = body' oo unwrap oo wrap
  by (simp add: cfun-eq-iff)
\end{verbatim}

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

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

\begin{verbatim}
lemma fix-body' \subseteq fix-(unwrap oo body oo wrap)
  by (rule fusion-partially-correct[OF wrap-unwrap refl body-body'])
\end{verbatim}

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

\begin{verbatim}
lemma \neg fix-(unwrap oo body oo wrap) \subseteq fix-body'
proof -
  have l: fix-(unwrap oo body oo wrap) = B \cdot A
    by (subst fix-eq) simp
  have r: fix-body' = \bot
    by (simp add: fix-strict)
  from l r show \?thesis by simp
qed
\end{verbatim}

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 \( \text{wrap} \circ \text{unwrap} = \text{ID} \), then making \( \text{unwrap} \) strict preserves this equation:

**lemma**

assumes \( \text{wrap} \circ \text{unwrap} = \text{ID} \)

shows \( \text{wrap} \circ \text{strictify} \cdot \text{unwrap} = \text{ID} \)

**proof** (rule cfun-eqI)

fix \( x \)

from \( \text{assms} \)

show \( (\text{wrap} \circ \text{strictify} \cdot \text{unwrap}) \cdot x = \text{ID} \cdot x \)

by (cases \( x = \bot \)) (simp-all add: cfun-eq-iff retraction-strict)

qed

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

### 4 A totally-correct fusion rule

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

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

**lemma** \textit{worker-wrapp-er-fusion-new}:

fixes \( \text{wrap} :: \mathcal{b} \rightarrow \mathcal{a} \)

fixes \( \text{unwrap} :: \mathcal{a} \rightarrow \mathcal{b} \)

fixes \( \text{body'} :: \mathcal{b} \rightarrow \mathcal{b} \)

assumes \( \text{wrap-unwrap}: \text{wrap} \circ \text{unwrap} = (\text{ID} :: \mathcal{a} \rightarrow \mathcal{a}) \)

assumes \( \text{unwrap-strict}: \text{unwrap} \cdot \bot = \bot \)

assumes \( \text{body-body'}: \text{unwrap} \circ \text{body} \circ \text{wrap} = \text{body'} \circ (\text{unwrap} \circ \text{unwrap}) \)

shows \( \text{fix-body} = \text{wrap} \cdot \text{fix-body'} \)

**proof**

from \( \text{body-body'} \)

have \( \text{unwrap} \circ \text{body} \circ (\text{wrap} \circ \text{unwrap}) = (\text{body'} \circ \text{unwrap} \circ (\text{unwrap} \circ \text{unwrap})) \)

by (simp add: assoc-oo)

with \( \text{wrap-unwrap} \) have \( \text{unwrap} \circ \text{body} \circ \text{body} = \text{body'} \circ \text{unwrap} \)

by simp

with \( \text{unwrap-strict} \) have \( \text{unwrap} \cdot (\text{fix-body}) = \text{fix-body'} \)

by (rule lfp-fusion)

hence \( (\text{wrap} \circ \text{unwrap}) \cdot (\text{fix-body}) = \text{wrap} \cdot (\text{fix-body'}) \)

by simp

with \( \text{wrap-unwrap} \) show \(?\text{thesis}\) by simp

qed
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:

**lemma** worker-wrapper-fusion-new-general:
  *fixes* wrap :: 'b::pcpo ⇒ 'a::pcpo
  *fixes* unwrap :: 'a ⇒ 'b
  *assumes* wrap-unwrap: wrap oo unwrap = (ID :: 'a ⇒ 'a)
  *assumes* unwrap-strict: unwrap⊥ = ⊥
  *assumes* body-body': ∀r. (unwrap oo wrap) r = r

shows fix-body = wrap·(fix-body')

**proof**
  let ?P = λ(x, y). x = y ∧ unwrap·(wrap·x) = x
  have ?P (fix·(unwrap oo body oo wrap), (fix·body'))
  **proof** (induct rule: parallel-fix-ind)
    **case** 2 with retraction-strict unwrap-strict wrap-unwrap
    show ?P (⊥, ⊥)
    by (bestsimp simp add: cfun-eq-iff)
    **case** (3 x y)
    hence xy: x = y and unwrap-wrap: unwrap·(wrap·x) = x by auto
    from body-body' xy unwrap-wrap
    have (unwrap oo body oo wrap)·x = body'·y
    by simp
    moreover
    from wrap-unwrap
    have unwrap·(wrap·((unwrap oo body oo wrap)·x)) = (unwrap oo body oo wrap)·x
    by (simp add: cfun-eq-iff)
    ultimately show ?case by simp
  qed simp
  thus ?thesis
  using worker-wrapper-id[OF wrap-unwrap refl] by simp
  qed

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 list ⇒ 'a list

definition
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:

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

(worker/wrapper fusion)

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

\[
\text{list2H} :: 'a \text{llist} \rightarrow 'a \text{H} \text{ where} \\
\text{list2H} \equiv \text{lappend}
\]

lemma acc-c2a-strict[simp]: \( \text{list2H} \perp = \perp \) 

by (rule cfun-eqI, simp add: list2H-def)

definition

\[
\text{H2list} :: 'a \text{H} \rightarrow 'a \text{llist} \text{ where} \\
\text{H2list} \equiv \Lambda \ f. \ f\text{-lnil}
\]

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 \( \perp = (\Lambda \ x. \ \perp) \), which does not hold in Haskell.

lemma \( \text{H-list-hom-append} \): \( \text{list2H} \cdot (xs :++ ys) = \text{list2H} \cdot xs \ oo \ \text{list2H} \cdot ys \) (is \( ?\text{lhs} = ?\text{rhs} \))

proof (rule cfun-eqI)

fix \ z s

have \( ?\text{lhs} \cdot zs = (xs :++ ys) :++ zs \) by (simp add: list2H-def)

also have \( \ldots = xs :++ (ys :++ zs) \) by (rule lappend-assoc)

also have \( \ldots = \text{list2H} \cdot xs \cdot (ys :++ zs) \) by (simp add: list2H-def)

also have \( \ldots = \text{list2H} \cdot xs \cdot (\text{list2H} \cdot ys \cdot zs) \) by (simp add: list2H-def)

also have \( \ldots = (\text{list2H} \cdot xs \ oo \ \text{list2H} \cdot ys) \cdot zs \) by simp

finally show \( ?\text{lhs} \cdot zs = (\text{list2H} \cdot xs \ oo \ \text{list2H} \cdot ys) \cdot zs \).

qed

lemma \( \text{H-list-hom-id} \): \( \text{list2H} \cdot \text{lnil} = \text{ID} \) by (simp add: list2H-def)

lemma \( \text{H2list-list2H-inv} \): \( \text{H2list} \ oo \ \text{list2H} = \text{ID} \)

by (rule cfun-eqI, simp add: H2list-def list2H-def)

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

fixrec \( \text{lrev} :: 'a \text{llist} \rightarrow 'a \text{llist} \)
where
\[ \text{lrev-lnil} = \text{lnil} \]
| \text{lrev}(x \@ xs) = \text{lrev} \cdot xs :++ (x \@ \text{lnil}) |

Note “body” is the generator of \text{lrev-def}.

**Lemma** \text{lrev-strict}: \text{lrev} \cdot \bot = \bot  
by \text{fixrec-simp}

**Fixrec** \text{lrev-body} :: ('a list \to 'a list) \to 'a list \to 'a list
where
\[ \text{lrev-body} \cdot r \cdot \text{lnil} = \text{lnil} \]
| \text{lrev-body} \cdot r \cdot (x :@ xs) = r \cdot xs :++ (x \@ \text{lnil})

**Lemma** \text{lrev-body-strict}: \text{lrev-body} \cdot r \cdot \bot = \bot  
by \text{fixrec-simp}

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} \cdot \text{lrev-body}  
by (rule \text{cfun-eqI}, subst \text{lrev-def}, subst \text{lrev-body.unfold, simp})

Wrap / unwrap functions.

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

**Lemma** \text{unwrapH-strict}: \text{unwrapH} \cdot \bot = \bot  
unfolding \text{unwrapH-def} by (rule \text{cfun-eqI, simp})

**Definition**  
\text{wrapH} :: ('a list \to 'a \to 'H) \to 'a list \to 'a list  
where  
\[ \text{wrapH} \equiv \Lambda f \cdot \text{H2list} \cdot (f \cdot \text{xs}) \]

**Lemma** \text{wrapH-unwrapH-id}: \text{wrapH} \circ \text{unwrapH} = \text{ID} (\text{is \ ?lhs = \ ?rhs})  
proof (rule \text{cfun-eqI})+  
fix f \term{xs}  
have \?lhs \cdot f \cdot \text{xs} = \text{H2list} \cdot \text{list2H} \cdot (f \cdot \text{xs}) by (simp add: \text{wrapH-def \ unwrapH-def})  
also have \ldots = \text{H2list} \circ \text{list2H} \cdot (f \cdot \text{xs}) by simp  
also have \ldots = \text{ID} \cdot (f \cdot \text{xs}) by (simp only: \text{H2list-list2H-inv})  
also have \ldots = \?rhs \cdot f \cdot \text{xs} by simp  
finally show \?lhs \cdot f \cdot \text{xs} = \?rhs \cdot f \cdot \text{xs} .

qed

5.2 Gill/Hutton-style worker/wrapper.

**Definition**  
\text{lrev-work} :: 'a list \to 'a \to 'H  
\text{lrev-work} \equiv \text{fix} \cdot (\text{unwrapH} \circ \text{lrev-body} \circ \text{wrapH})
definition
lrev-wrap : 'a llist → 'a llist where
lrev-wrap ≡ wrapH-lrev-work

lemma lrev-lrev-ww-eq: lrev = lrev-wrap
using worker-wrapper-id[OF wrapH-unwrapH-id lrev-lrev-body-eq]
by (simp add: lrev-wrap-def lrev-work-def)

5.3 Optimise worker/wrapper.

Intermediate worker.

fixrec lrev-body1 :: ('a llist → 'a H) → 'a llist → 'a H where
lrev-body1·r·lnil = list2H·lnil
| lrev-body1·r·(x :@ xs) = list2H·(wrapH·r·xs ++ (x :@ lnil))

definition
lrev-work1 :: 'a llist → 'a H where
lrev-work1 ≡ fix-lrev-body1

lemma lrev-body-lrev-body1-eq: lrev-body1 = unwrapH oo lrev-body oo wrapH
apply (rule cfun-eqI)+
apply (subst lrev-body)
apply (unfold)
apply (case-tac xa)
apply (simp-all add: list2H-def wrapH-def unwrapH-def)
done

lemma lrev-work1-lrev-work-eq: lrev-work1 = lrev-work
by (unfold lrev-work-def lrev-work1-def, rule cfun-arg-cong[OF lrev-body-lrev-body1-eq])

Now use the homomorphism.

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

lemma lrev-body2-strict[simp]: lrev-body2·⊥ = ⊥
by fixrec-simp

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

lemma lrev-work2-strict[simp]: lrev-work2·⊥ = ⊥
unfolding lrev-work2-def
by (subst fix-eq) simp

13
lemma \text{lrev-body2-lrev-body1-eq}: \text{lrev-body2} = \text{lrev-body1} \\
\text{by} \ ((\text{rule cfun-eqI})+ \\
, (\text{subst \text{lrev-body1}.unfold}, \text{subst \text{lrev-body2}.unfold}) \\
, (\text{simp add: H-list-hom-append[symmetric] H-list-hom-id}))

lemma \text{lrev-work2-lrev-work1-eq}: \text{lrev-work2} = \text{lrev-work1} \\
\text{by} \ ((\text{unfold \text{lrev-work2-def} \text{lrev-work1-def}}) \\
, \text{rule cfun-arg-cong[OF \text{lrev-body2-lrev-body1-eq}])}

Simplify.

fixrec \text{lrev-body3} :: ('a llist \rightarrow 'a H) \rightarrow 'a H llist \rightarrow 'a H \\
where \text{lrev-body3} \cdot r \cdot \text{lnil} = \text{ID} \\
| \text{lrev-body3} \cdot r \cdot (x :@ xs) = r \cdot xs oo \text{list2H} \cdot (x :@ \text{lnil})

lemma \text{lrev-body3-strict[simp]}: \text{lrev-body3} \cdot r \cdot \bot = \bot \\
\text{by} \ \text{fixrec-simp}

definition \text{lrev-work3} :: 'a llist \rightarrow 'a H where \\
\text{lrev-work3} \equiv \text{fix} \cdot \text{lrev-body3}

lemma \text{lrev-wwfusion}: \text{list2H} \cdot ((\text{wrapH} \cdot \text{lrev-work2}) \cdot \text{xs}) = \text{lrev-work2} \cdot \text{xs} \\
\text{proof} \ − \ \\
\{ \\
\begin{aligned}
\text{have list2H oo \text{wrapH} \cdot \text{lrev-work2} = \text{unwrapH} \cdot \text{wrapH} \cdot \text{lrev-work2}} \\
&\text{by} \ (\text{rule cfun-eqI}, \ \text{simp add: \text{unwrapH-def}}) \\
&\text{also have \ldots = (\text{unwrapH oo \text{wrapH}}) \cdot \text{lrev-work2} \ \text{by simp}} \\
&\text{also have \ldots = \text{lrev-work2}} \\
&\text{apply −} \\
&\text{apply (\text{rule worker-wrapper-fusion[OF \text{wrapH-unwrapH-id}, \text{where body=\text{lrev-body3}]})}} \\
&\text{apply (auto iff: \text{lrev-body2-lrev-body1-eq \text{lrev-body3-lrev-body1-eq \text{lrev-work2-def \text{lrev-work1-def})}}}} \\
&\text{done} \\
&\text{finally have list2H oo \text{wrapH} \cdot \text{lrev-work2} = \text{lrev-work2} .} \\
\end{aligned}
\} \\
\text{thus \?thesis using cfun-eq-iff[where \ f=list2H oo \text{wrapH} \cdot \text{lrev-work2} \text{ and g=\text{lrev-work2}]} \ \\
\text{by auto} \\
\text{qed}

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

lemma \text{lrev-work3} \sqsubseteq \text{lrev-work2} \\
\text{unfolding \text{lrev-work3-def}}
\text{proof}(\text{rule fix-least}) \\
\{ \\
\begin{aligned}
&\text{fix \text{xs have \text{lrev-body3} \cdot \text{lrev-work2}} \cdot \text{xs} = \text{lrev-work2} \cdot \text{xs}} \\
&\text{proof(cases \text{xs})} \\
&\quad \text{case \text{bottom thus \?thesis by simp} \\
\end{aligned}
\}
next

case lnil thus \(?thesis
  unfolding lrev-work2-def
  by (subst fix-eq[where F=lrev-body2, simp)
next

case (\(\text{lcons}\ y\ zs\))
  hence \(\text{lrev-body3} \cdot \text{lrev-work2} \cdot \text{xs} = \text{lrev-work2} \cdot \text{ys} oo \text{list2H}(y \cdot \text{nil})\) by simp
  also have \(\ldots = \text{list2H}((\text{wrapH}\cdot \text{lrev-work2})\cdot \text{ys}) oo \text{list2H}(y \cdot \text{nil})\)
    using lrev-wwfusion[where xs=ys] by simp
  also from lcons have \(\ldots = \text{lrev-body2} \cdot \text{lrev-work2} \cdot \text{xs}\) by simp
  also have \(\ldots = \text{lrev-work2} \cdot \text{xs}\)
    unfolding lrev-work2-def by (simp only: fix-eq[symmetric])
  finally show ?thesis by simp
qed

\(\text{lrev-body3} \cdot \text{lrev-work2} = \text{lrev-work2}\) by (rule cfun-eqI)

qed

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 com-
pletely sound in this context, by appealing to the lazy list induction prin-
ciple.

\textbf{Lemma} \(\text{lrev-work3} \cdot \text{lrev-work2} = \text{lrev-work2}\) \(\text{(is \?lhs = \?rhs)}\)

\textbf{Proof} (rule cfun-eqI)

\textbf{Fix} \(x\)

\textbf{Show} \?lhs-\(?x\) = \?rhs-\(?x\)

\textbf{Proof} (induct \(?x\))

\textbf{Show} \(\text{lrev-work3} \cdot \perp = \text{lrev-work2} \cdot \perp\)
  \textbf{Apply} (unfold lrev-work3-def lrev-work2-def)
  \textbf{Apply} (subst fix-eq[where F=lrev-body2])
  \textbf{Apply} (subst fix-eq[where F=lrev-body3])
  \textbf{By} \(\text{simp add: lrev-body3.unfold lrev-body2.unfold}\)

\textbf{Next}

\textbf{Show} \(\text{lrev-work3} \cdot \text{lnil} = \text{lrev-work2} \cdot \text{lnil}\)
  \textbf{Apply} (unfold lrev-work3-def lrev-work2-def)
  \textbf{Apply} (subst fix-eq[where F=lrev-body2])
  \textbf{Apply} (subst fix-eq[where F=lrev-body3])
  \textbf{By} \(\text{simp add: lrev-body3.unfold lrev-body2.unfold}\)

\textbf{Next}

\textbf{Fix} \(a\ l\ \text{assume} \ lrev-work3 \cdot \perp = \text{lrev-work2} \cdot \perp\)

\textbf{Thus} \(\text{lrev-work3} - (a \cdot \text{lnil}) = \text{lrev-work2} - (a \cdot \text{lnil})\)
  \textbf{Apply} (unfold lrev-work3-def lrev-work2-def)
  \textbf{Apply} (subst fix-eq[where F=lrev-body2])
  \textbf{Apply} (subst fix-eq[where F=lrev-body3])
apply (fold lrev-work3-def lrev-work2-def)
apply (simp add: lrev-body3. unfold lrev-body2. unfold lrev-wwfusion)
done
qed simp-all
qed

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

lemma lrev3-2-syntactic: lrev-body3 oo (unwrapH oo wrapH) = lrev-body2
apply (subst lrev-body2.unfold, subst lrev-body3.unfold)
apply (rule cfun-eqI)+
apply (case-tac xa)
apply (simp-all add: unwrapH-def)
done

lemma lrev-work3-lrev-work2-eq': lrev = wrapH·lrev-work3
proof -
  from lrev-lrev-body-eq
  have lrev = fix·lrev-body .
  also from wrapH-unwrapH-id unwrapH-strict
  have \ldots = wrapH·(fix·lrev-body3)
    by (rule worker-wrapper-fusion-new
      , simp add: lrev3-2-syntactic lrev-body2-lrev-body1-eq lrev-body-lrev-body1-eq)
  finally show \?thesis unfolding lrev-work3-def by simp
qed

Final syntactic tidy-up.

fixrec lrev-body-final :: ('a llist → 'a H) → 'a llist → 'a H
where
  lrev-body-final·r·lnil·ys = ys
| lrev-body-final·r·(x :@ xs)·ys = r·xs·(x :@ ys)

definition
  lrev-work-final :: 'a llist → 'a H where
    lrev-work-final ≡ fix·lrev-body-final

definition
  lrev-final :: 'a llist → 'a llist where
    lrev-final ≡ \Lambda xs. lrev-work-final·xs·lnil

lemma lrev-body-final-lrev-body3-eq': lrev-body-final·r·xs = lrev-body3·r·xs
apply (subst lrev-body-final.unfold)
apply (subst lrev-body3.unfold)
apply (cases xs)
apply (simp-all add: list2H-def ID-def cfun-eqI)
done

lemma lrev-body-final-lrev-body3-eq: lrev-body-final = lrev-body3
by (simp only: lrev-body-final-lrev-body3-eq' cfun-eqI)
lemma lrev-final-lrev-eq: lrev = lrev-final (is ?lhs = ?rhs)
proof −
  have ?lhs = lrev-wrap by (rule lrev-lrev-ww-eq)
  also have ... = wrapH·lrev-work by (simp only: lrev-wrap-def)
  also have ... = wrapH·lrev-work1 by (simp only: lrev-work1-lrev-work-eq)
  also have ... = wrapH·lrev-work2 by (simp only: lrev-work2-lrev-work1-eq)
  also have ... = wrapH·lrev-work3 by (simp only: lrev-work3-lrev-work2-eq)
  also have ... = wrapH·lrev-work-final by (simp only: lrev-work3-def lrev-work-final-def
lrev-body-final-lrev-body3-eq)
  also have ... = lrev-final by (simp add: lrev-final-def cfun-eqI H2list-def wrapH-def)
  finally show ?thesis .
qed

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 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 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 → Nat
where
  fac·n = If n =B 0 then 1 else n · fac·(n − 1)
declare fac.simps[simp del]

lemma fac-strict[simp]: fac·⊥ = ⊥
by fixrec-simp

definition fac-body :: (Nat → Nat) → Nat → Nat where
fac-body ≡ Λ r n. If n =B 0 then 1 else n · r·(n − 1)

lemma fac-body-strict[simp]: fac-body·r·⊥ = ⊥
unfolding fac-body-def by simp
lemma fac-fac-body-eq: fac = fix fac-body
  unfolding fac-body-def by (rule cfun-eqI, subst fac-def, simp)

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 → Nat) → UNat → UNat⊥ where
  unwrapB ≡ Λ f. unbox oo f oo box

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

definition
  wrapB :: (UNat → UNat⊥) → Nat → Nat where
  wrapB ≡ Λ f x . unbox·x >>= f >>= box

lemma wrapB-unwrapB-body:
  assumes strictF: f·⊥ = ⊥
  shows (wrapB oo unwrapB)·f = f (is ?lhs = ?rhs)

proof (rule cfun-eqI)
  fix x :: Nat
  have ?lhs·x = unbox·x >>= (Λ x′. unwrapB·f·x′ >>= box)
    unfolding wrapB-def by simp
  also have ... = unbox·x >>= (Λ x′. unbox·(f·(box·x′))) >>= box
    unfolding unwrapB-def by simp
  also from strictF have ... = f·x by (cases x, simp-all)
  finally show ?lhs·x = ?rhs·x.

qed

Apply worker/wrapper.

definition
  fac-work :: UNat → UNat⊥ where
  fac-work ≡ fix (unwrapB oo fac-body oo wrapB)

definition
  fac-wrap :: Nat → Nat where
  fac-wrap ≡ wrapB·fac-work

lemma fac-fac-ww-eq: fac = fac-wrap (is ?lhs = ?rhs)
  proof –
    have wrapB oo unwrapB oo fac-body = fac-body
      using wrapB-unwrapB-body[OF fac-body-strict]
      by – (rule cfun-eqI, simp)
    thus ?thesis
      using worker-wrapper-body[where computation=fac and body=fac-body and
                                wrap=wrapB and unwrap=unwrapB]
unfolding fac-work-def fac-wrap-def by (simp add: fac-fac-body-eq)

qed

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

definition fac-body' :: (UNat → UNat) → UNat → UNat where
  fac-body' ≡ Λ r n.
  unbox·(If box·n =B 0
  then 1
  else unbox·(box·n - 1) >>= r >>= (Λ b. box·n * box·b)))

lemma fac-body'-fac-body: fac-body' = unwrapB oo fac-body oo wrapB (is ?lhs = ?rhs)
proof (rule cfun-eqI)+
  fix r x
  show ?lhs·r·x = ?rhs·r·x
  using bbind-case-distr-strict [where f =Λ y. box·x * y and g = unbox·(box·x - 1)]
  unfolding fac-body-final-def fac-body-def unwrapB-def wrapB-def by simp
qed

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 → UNat) → UNat → UNat where
  fac-body-final ≡ Λ r n.
  if n = 0 then up·1 else r·(n - # 1) >>= (Λ b. up·(n *# b))

lemma fac-body-final-fac-body': fac-body-final = fac-body' (is ?lhs = ?rhs)
proof (rule cfun-eqI)+
  fix r x
  show ?lhs·r·x = ?rhs·r·x
  using bbind-case-distr-strict [where f = unbox and g = r·(x - # 1) and h = (Λ b. box·(x *# b))]
  unfolding fac-body-final-def fac-body'-def uMinus-def uMult-def zero-Nat-def one-Nat-def
  by simp
qed

definition fac-work-final :: UNat → UNat where
  fac-work-final ≡ fix·fac-body-final

definition fac-final :: Nat → Nat where
  fac-final ≡ Λ n. unbox·n >>= fac-work-final >>= box
lemma fac-fac-final: fac = fac-final (is \( ?lhs = ?rhs \))
proof -
  have \( ?lhs = \text{fac-wrap} \) by (rule fac-fac-ww-eq)
  also have \( \ldots = \text{wrapB-fac-work} \) by (simp only: fac-wrap-def)
  also have \( \ldots = \text{wrapB-fac-work-final} \) by (simp only: fac-work-final-def)
  also have \( \ldots = \text{fac-final} \) by (simp add: fac-final-def wrapB-def)
  finally show \( \text{thesis} \).
qed

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 \( \rightarrow \) UNat\( \bot \)

**definition**
\[ n2a :: UNat \rightarrow UNatAcc \text{ where} \]
\[ n2a \equiv \Lambda m n. \text{up}(m \,*\#\, n) \]

**definition**
\[ a2n :: UNatAcc \rightarrow UNat\,\bot \text{ where} \]
\[ a2n \equiv \Lambda a. a\cdot 1 \]

**lemma** \( a2n\)-strict\( [\text{simp}] \): \( a2n\,\bot = \bot \)
unfolding \( a2n\)-def \text{ by simp}

**lemma** \( a2n\,-\,n2a\): \( a2n\,(n2a\cdot u) = \text{up}\cdot u \)
unfolding \( \text{a2n-def n2a-def} \text{ by (simp add: uMult-arithmetic)} \)

**lemma** A-hom-mult: \( n2a\cdot(x \,*\#\, y) = (n2a\cdot x \gg \gg n2a\cdot y) \)
unfolding \( \text{n2a-def bKleisli-def by (simp add: uMult-arithmetic)} \)

**definition**
\[ \text{unwrapA} :: (UNat \rightarrow \text{UNat}_\bot) \rightarrow UNat \rightarrow \text{UNatAcc} \text{ where} \]
\[ \text{unwrapA} \equiv \Lambda f n. f\cdot n \gg \gg n2a \]

**lemma** unwrapA-strict\( [\text{simp}] \): unwrapA\( \bot = \bot \)
unfolding unwrapA-def \text{ by (rule cfun-eqI) simp}

20
definition
\(\text{wrapA} :: (\text{UNat} \to \text{UNatAcc}) \to \text{UNat} \to \text{UNat} \downarrow\) where
\(\text{wrapA} \equiv \Lambda f \cdot a2n \circ f\)

lemma \(\text{wrapA} \circ \text{unwrapA} = \text{ID}\)

unfolding \(\text{wrapA-def} \ \text{unwrapA-def}\)
apply (\text{rule cfun-eqI})+
apply (\text{case-tac} \ x \cdot xa)
apply (\text{simp-all add:} \ a2n-n2a)
done

Some steps along the way.

definition
\(\text{fac-acc-body1} :: (\text{UNat} \to \text{UNatAcc}) \to \text{UNat} \to \text{UNatAcc} \) where
\(\text{fac-acc-body1} \equiv \Lambda r \ n \cdot \begin{cases} \text{if} \ n = 0 \ \text{then} \ n2a \cdot 1 \ \text{else} \ \text{wrapA} \cdot r \cdot (n - \# 1) \gg= (\Lambda \ res. \ n2a \cdot (n * \# \ res)) \end{cases}\)

lemma \(\text{fac-acc-body1-fac-body-final-eq}\):
\(\text{fac-acc-body1} = \text{unwrapA} \circ \text{fac-body-final} \circ \text{wrapA}\)

unfolding \(\text{fac-acc-body1-def} \ \text{fac-body-final-def} \ \text{wrapA-def} \ \text{unwrapA-def}\)
by (\text{rule cfun-eqI})+ \text{simp}

Use the homomorphism.

definition
\(\text{fac-acc-body2} :: (\text{UNat} \to \text{UNatAcc}) \to \text{UNat} \to \text{UNatAcc} \) where
\(\text{fac-acc-body2} \equiv \Lambda r \ n \cdot \begin{cases} \text{if} \ n = 0 \ \text{then} \ n2a \cdot 1 \ \text{else} \ \text{wrapA} \cdot r \cdot (n - \# 1) \gg= (\Lambda \ res. \ n2a \cdot (n * \# \ res)) \end{cases}\)

lemma \(\text{fac-acc-body2-body1-eq}\):
\(\text{fac-acc-body2} \circ \text{wrapA} \circ \text{unwrapA} = \text{fac-acc-body1}\)

unfolding \(\text{fac-acc-body2-def} \ \text{wrapA-def} \ \text{unwrapA-def}\)
by (\text{rule cfun-eqI})+ (\text{simp add: A-hom-mult})

Apply worker/wrapper.

definition
\(\text{fac-acc-body3} :: (\text{UNat} \to \text{UNatAcc}) \to \text{UNat} \to \text{UNatAcc} \) where
\(\text{fac-acc-body3} \equiv \Lambda r \ n \cdot \begin{cases} \text{if} \ n = 0 \ \text{then} \ n2a \cdot 1 \ \text{else} \ n2a \cdot n \gg= r \cdot (n - \# 1) \end{cases}\)

lemma \(\text{fac-acc-body3-body2}\):
\(\text{fac-acc-body3} \circ \text{unwrapA} \circ \text{wrapA} = \text{fac-acc-body2}\)
(is ?lhs=?rhs)

proof (\text{rule cfun-eqI})+
fix \(r \ n \ acc\)
show ((\text{fac-acc-body3} \circ \text{unwrapA} \circ \text{wrapA}) \cdot r \cdot n \cdot acc) = \text{fac-acc-body2} \cdot r \cdot n \cdot acc
unfolding \(\text{fac-acc-body2-def} \ \text{fac-acc-body3-def} \ \text{wrapA-def}\)
using \(\text{bbind-case-distr-strict}\) [where \(f = \Lambda y. \ n2a \cdot n \gg= y\) and \(h = n2a, \ \text{symmetric}\)]
by (\text{simp})

qed
lemma fac-work-final-body3-eq: fac-work-final = wrapA·(fix·fac-acc-body3)
unfolding fac-work-final-def
by (rule worker-wrapper-fusion-new[OF wrapA-unwrapA-id unwrapA-strict])
  (simp add: fac-acc-body3-body2 fac-acc-body2-body1-eq fac-acc-body1-fac-body-final-eq)

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)
unfolding fac-acc-body3-def fac-acc-body-final-def n2a-def bKleisli-def
by (rule cfun-eqI)+
  (simp add: uMult-arithmetic)

lemma fac-acc-work-final-fac-work: fac-acc-work-final = fac-work-final (is ?lhs=?rhs)
proof –
  have ?rhs = wrapA·(fix·fac-acc-body3) by (rule fac-work-final-body3-eq)
  also have . . . = wrapA·(fix·fac-acc-body-final)
    using fac-acc-work-final-fac-acc-work3-eq by simp
  also have . . . = ?lhs
    unfolding fac-acc-work-final-def wrapA-def a2n-def
    by (simp add: cfun-comp1)
  finally show ?thesis by simp
qed

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)

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

lemma smap-smap: smap·f·(smap·g·xs) = smap·(f oo g)·xs
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

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
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 (rule cfun-eqI)
  fix f show ?lhs·f = ?rhs·f
  proof (coinduct rule: Stream.coinduct)
    let ?R = λs s'. (∃ f. s = f·0 & & unwrapS·(f oo (Λ x. 1 + x))
    ∧ s' = f·0 & & smap·(f oo (Λ x. 1 + x))·nats)
    show Stream-bisim ?R
    proof
      fix s s' assume ?R s s'
      then obtain f where fs: s = f·0 & & unwrapS·(f oo (Λ x. 1 + x))
      and fs': s' = f·0 & & smap·(f oo (Λ x. 1 + x))·nats
      by blast
      have ?R (unwrapS·(f oo (Λ x. 1 + x))) (smap·(f oo (Λ x. 1 + x))·nats)
      by ( rule ext[where x=f oo (Λ x. 1 + x)]
        , subst unwrapS·.unfold, subst nats.unfold, simp add: smap-smap)
      with fs fs'
      show (s = ⊥ & & s' = ⊥)
      ∨ (∃ h t t'.
        (∃ f. t = f·0 & & unwrapS·(f oo (Λ x. 1 + x))
        ∧ t' = f·0 & & smap·(f oo (Λ x. 1 + x))·nats)
        ∧ s = h & & t ∧ s' = h & & t' ) by best
    qed
show \( R \ (\text{lhs} \cdot f) \ (\text{rhs} \cdot f) \)

proof
- have lhs: \( \text{lhs} \cdot f = f \cdot 0 \) \&\& \( \text{unwrapS} \cdot (f \circ (\Lambda x. 1 + x)) \) by (subst \text{unwrapS}.unfold, simp)
  have rhs: \( \text{rhs} \cdot f = f \cdot 0 \) \&\& \( \text{smap} \cdot (f \circ (\Lambda x. 1 + x)) \cdot \text{nats} \) by (rule \text{unwrapS′}.unfold)
  from lhs rhs show \( \text{thesis} \) by best
qed

definition wrapS :: \( 'a \ 	ext{Stream} \to \text{Nat} \to 'a \) where
wrapS \equiv \( \Lambda s \ i. s \!! i \)

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 argument.

lemma wrapS-unwrapS-id':
  assumes strictF: \( f : \text{Nat} \to 'a \cdot \_ = \_ \)
  shows \( \text{unwrapS} \cdot f !! n = f \cdot n \)
using strictF
proof(induct n arbitrary: \( f \) rule: \text{Nat-induct})
  case bottom with strictF show \( \_ \) by simp
next
case zero thus \( \_ \) by (subst \text{unwrapS}.unfold, simp)
next
case (Suc i f)
  have \( \text{unwrapS} \cdot f !! (i + 1) = (f \cdot 0 \) \&\& \( \text{unwrapS} \cdot (f \circ (\Lambda x. 1 + x)) \) !! (i + 1)
    by (subst \text{unwrapS}.unfold, simp)
  also from Suc have \( \ldots = \text{unwrapS} \cdot (f \circ (\Lambda x. 1 + x)) \) !! i by simp
  also from Suc have \( \ldots = (f \circ (\Lambda x. 1 + x)) \cdot i \) by simp
  also have \( \ldots = f \cdot (i + 1) \) by (simp add: plus-commute)
  finally show \( \_ \) .
qed

lemma wrapS-unwrapS-id: \( f \cdot \_ = \_ \Rightarrow (\text{unwrapS} \circ \text{unwrapS}) \cdot f = f \)
  by (rule cfun-eqI, simp add: wrapS-unwrapS-id' \text{wrapS-def})

7.3 Fibonacci example.

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

definition fib :: \( \text{Nat} \to \text{Nat} \) where
fib \equiv \text{fix} \cdot \text{fib-body}
Apply worker/wrapper.

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

definition
  fib-wrap :: Nat → Nat where
  fib-wrap ≡ wrapS \(\cdot\) fib-work

lemma unwrapS-unwrapS-fib-body: unwrapS oo unwrapS oo fib-body = fib-body
  proof
  (rule cfun-eqI)
  fix r show (unwrapS oo unwrapS oo fib-body)\(r\) = fib-body\(r\)
  using unwrapS-unwrapS-id[where \(f=\)fib-body\(r\)] by simp
  qed

lemma fib-ww-eq: fib = fib-wrap
  using worker-wwraper-body[OF wrapS-unwrapS-fib-body]
  by (simp add: fib-def fib-wrap-def fib-work-def)

Optimise.

fixrec
  fib-work-final :: Nat Stream
and
  fib-f-final :: Nat → Nat
where
  fib-work-final = smap \(\cdot\) fib-f-final \(\cdot\) 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

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)
  proof
  let \(?wb = \Lambda r. Nat-case 1 (Nat-case 1 (Λ n’. r !! n’ + r !! (n’ + 1))))
  let \(?mr = \Lambda (juf :: Nat Stream, jff). (smap jff-nats, \(?wb \cdot juf\))
  have \(?lhs = \fst (fix \cdot\?mr)
  by (simp add: fib-work-final-def split-def csplit-def)

\(^1\)The interested reader can find some historical commentary in Harel (1980); Sangiorgi (2009).
also have ... = (µ f w f, fst (?mr·(f w f, µ f f, snd (?mr·(f w f, f f)))))
  using fix-cprod[where F = ?mr] by simp
also have ... = (µ f w f, smap (µ f f · (?w b·f w f) · nats) by simp
also have ... = (µ f w f, smap (?w b·f w f) · nats) by (simp add: fix-const)
also have ... = ?rhs
unfolding fib-body-def fib-work-def unwrapS-unwrapS'-eq unwrapS'-def wrapS-def
by (simp add: cfcomp1)
finally show ?thesis .
qed

lemma fib-final-fib-eq: fib-final = fib (is ?lhs = ?rhs)
proof –
  have ?lhs = (Λ n. fib-work-final !! n)
  by (simp add: fib-final-def)
also have ... = (Λ n. fib-work !! n)
  by (simp only: fib-work-final-fib-work-eq)
also have ... = fib-wrap
  by (simp add: fib-wrap-def wrapS-def)
also have ... = ?rhs
  by (simp only: fib-ww-eq)
finally show ?thesis .
qed

8 Tagless interpreter via double-barreled continuations

\textbf{type-synonym} ‘a Cont = (‘a \rightarrow ‘a) \rightarrow ‘a

\textbf{definition}
val2cont :: ‘a \rightarrow ‘a Cont
val2cont ≡ (Λ a c. c·a)

\textbf{definition}
cont2val :: ‘a Cont \rightarrow ‘a
cont2val ≡ (Λ f. f·ID)

\textbf{lemma} cont2val-val2cont-id: cont2val oo val2cont = ID
by (rule cfun-eqI, simp add: val2cont-def cont2val-def)

\textbf{domain} Expr =
\begin{itemize}
\item Val (lazy val::Nat)
\item Add (lazy addl::Expr) (lazy addr::Expr)
\item Throw
\item Catch (lazy cbody::Expr) (lazy Chandler::Expr)
\end{itemize}

\textbf{fixrec} eval :: Expr \rightarrow Nat Maybe
\textbf{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)

26
\begin{align*}
\text{fixrec} \quad \text{eval-body} &:: (\text{Expr} \to \text{Nat Maybe}) \to \text{Expr} \to \text{Nat Maybe} \\
\text{where} &
\begin{align*}
\text{eval-body} \cdot r \cdot (\text{Val} \cdot n) &= \text{Just} \cdot n \\
\text{eval-body} \cdot r \cdot (\text{Add} \cdot x \cdot y) &= \text{mliftM2} (\Lambda \, a \, b \, : \, a + b) \cdot (r \cdot x) \cdot (r \cdot y) \\
\text{eval-body} \cdot r \cdot \text{Throw} &= \text{mfail} \\
\text{eval-body} \cdot r \cdot (\text{Catch} \cdot x \cdot y) &= \text{mcatch} \cdot (r \cdot x) \cdot (r \cdot y)
\end{align*}
\end{align*}

\textbf{lemma} \quad \text{eval-body-strictExpr[simp]}: \text{eval-body} \cdot r \cdot \bot = \bot \\
\text{by (subst eval-body, unfold, simp)}

\textbf{lemma} \quad \text{eval-eval-body-eq}: \text{eval} = \text{fix} \cdot \text{eval-body} \\
\text{by (rule cfun-eqI, subst eval-def, subst eval-body, unfold, simp)}

\subsection{Worker/wrapper}

\textbf{definition} \quad \text{unwrapC} :: (\text{Expr} \to \text{Nat Maybe}) \to (\text{Expr} \to (\text{Nat} \to \text{Nat Maybe}) \to \text{Nat Maybe} \\
\to \text{Nat Maybe}) \quad \text{where} \\
\text{unwrapC} \equiv \Lambda \, g \, e \, s \, f. \, \text{case} \, e \, \text{of} \, \text{Nothing} \Rightarrow f \mid \text{Just-}n \Rightarrow s \cdot n

\textbf{lemma} \quad \text{unwrapC-strict[simp]}: \text{unwrapC} \cdot \bot = \bot \\
\text{unfolding} \quad \text{unwrapC-def by (rule cfun-eqI) + simp}

\textbf{definition} \quad \text{wrapC} :: (\text{Expr} \to (\text{Nat} \to \text{Nat Maybe}) \to \text{Nat Maybe} \\
\to \text{Nat Maybe}) \quad \text{where} \\
\text{wrapC} \equiv \Lambda \, g \, e \cdot \text{Just} \cdot \text{Nothing}

\textbf{lemma} \quad \text{wrapC-unwrapC-id}: \text{wrapC} \circ \text{unwrapC} = \text{ID} \\
\text{proof} (\text{intro cfun-eqI}) \\
\text{fix} \, g \, e \\
\text{show} \, (\text{wrapC} \circ \text{unwrapC}) \cdot g \cdot e = \text{ID} \cdot g \cdot e \\
\text{by (cases g \cdot e, simp-all add: wrapC-def unwrapC-def)}
\text{qed}

\textbf{definition} \quad \text{eval-work} :: \text{Expr} \to (\text{Nat} \to \text{Nat Maybe}) \to \text{Nat Maybe} \\
\text{eval-work} \equiv \text{fix} \cdot (\text{unwrapC} \circ \text{eval-body} \circ \text{wrapC})

\textbf{definition} \quad \text{eval-wrap} :: \text{Expr} \to \text{Nat Maybe} \quad \text{where} \\
\text{eval-wrap} \equiv \text{wrapC} \cdot \text{eval-work}

\textbf{fixrec} \quad \text{eval-body}' :: (\text{Expr} \to (\text{Nat} \to \text{Nat Maybe}) \to \text{Nat Maybe} \to \text{Nat Maybe} \\
\to \text{Expr} \to (\text{Nat} \to \text{Nat Maybe}) \to \text{Nat Maybe} \to \text{Nat Maybe}) \quad \text{where} \\
\text{eval-body}' \cdot r \cdot (\text{Val} \cdot n) \cdot s \cdot f &= s \cdot n \\
\text{eval-body}' \cdot r \cdot (\text{Add} \cdot x \cdot y) \cdot s \cdot f &= (\text{case} \, \text{wrapC} \cdot r \cdot x \, \text{of} 

\begin{align*}
\text{Nothing} & \Rightarrow f \\
\text{Just} \cdot n & \Rightarrow (\text{case wrapC} \cdot r \cdot y \ of \ \text{Nothing} \Rightarrow f \\
& \quad \mid \text{Just} \cdot m \Rightarrow s \cdot (n + m) )
\end{align*}

\begin{align*}
&\text{eval-body}' \cdot r \cdot \text{Throw} \cdot s \cdot f = f \\
&\text{eval-body}' \cdot r \cdot (\text{Catch} \cdot x \cdot y) \cdot s \cdot f = (\text{case wrapC} \cdot r \cdot x \ of \ \text{Nothing} \Rightarrow (\text{case wrapC} \cdot r \cdot y \ of \ \text{Nothing} \Rightarrow f \\
& \quad \mid \text{Just} \cdot n \Rightarrow s \cdot n ) \\
& \mid \text{Just} \cdot n \Rightarrow s \cdot n )
\end{align*}

\textbf{lemma eval-body}'-strictExpr[simp]: eval-body}' \cdot \bot \cdot s \cdot f = \bot
\text{by (subst eval-body}', unfold, simp)}

\textbf{definition eval-work}': \text{Expr} \rightarrow (\text{Nat} \rightarrow \text{Nat Maybe}) \rightarrow \text{Nat Maybe} \rightarrow \text{Nat Maybe} \text{ where eval-work}'} \equiv \text{fix} \cdot \text{eval-body}'}

This proof is unfortunately quite messy, due to the simplifier’s inability to cope with HOLCF’s case distinctions.

\textbf{lemma eval-body}'-eval-body-eq: eval-body}' = \text{unwrapC} \ oo \ \text{eval-body} oo \ \text{wrapC}
\text{apply (intro cfun-eqI)}
\text{apply (unfold unwrapC-def wrapC-def)}
\text{apply (case-tac xa)}
\text{apply simp-all}
\text{apply (simp add: wrapC-def)}
\text{apply (case-tac x Expr1 \cdot Just \cdot Nothing)}
\text{apply simp-all}
\text{apply (case-tac x Expr2 \cdot Just \cdot Nothing)}
\text{apply simp-all}
\text{apply (simp add: mfail-def)}
\text{apply (simp add: mcatch-def wrapC-def)}
\text{apply (case-tac x Expr1 \cdot Just \cdot Nothing)}
\text{apply simp-all}
done

\textbf{fixrec eval-body-final} :: (\text{Expr} \rightarrow (\text{Nat} \rightarrow \text{Nat Maybe}) \rightarrow \text{Nat Maybe} \rightarrow \text{Nat Maybe}) \\
\rightarrow \text{Expr} \rightarrow (\text{Nat} \rightarrow \text{Nat Maybe}) \rightarrow \text{Nat Maybe} \rightarrow \text{Nat Maybe}
\text{where}
\text{eval-body-final} \cdot r \cdot (\text{Val} \cdot n) \cdot s \cdot f = s \cdot n \\
\text{eval-body-final} \cdot r \cdot (\text{Add} \cdot x \cdot y) \cdot s \cdot f = r \cdot x \cdot (\Lambda \ n. \ r \cdot y \cdot (\Lambda \ m. \ s \cdot (n + m)) \cdot f) \cdot f \\
\text{eval-body-final} \cdot r \cdot \text{Throw} \cdot s \cdot f = f \\
\text{eval-body-final} \cdot r \cdot (\text{Catch} \cdot x \cdot y) \cdot s \cdot f = r \cdot x \cdot s \cdot (r \cdot y \cdot s \cdot f)

\textbf{lemma eval-body-final-strictExpr[simp]: eval-body-final} \cdot r \cdot \bot \cdot s \cdot f = \bot
\text{by (subst eval-body-final, unfold, simp)}

\textbf{lemma eval-body}'-eval-body-final-eq: eval-body-final oo \ \text{unwrapC} oo \ \text{wrapC} = \text{eval-body}'
\text{apply (rule cfun-eqI)}+
apply (case-tac xa)
  apply (simp-all add: unwrapC-def)
done

definition
  eval-work-final :: Expr → (Nat → Nat Maybe) → Nat Maybe → Nat Maybe
where
  eval-work-final ≡ fix·eval-body-final

definition
  eval-final :: Expr → Nat Maybe
where
  eval-final ≡ (Λ e. eval-work-final·e·Just·Nothing)

lemma eval = eval-final
proof –
  have eval = fix·eval-body by (rule eval-eval-body-eq)
also from wrapC-unwrapC-id unwrapC-strict have . . . = wrapC·(fix·eval-body-final)
  apply (rule worker-wrapper-fusion-new)
  using eval-body'·eval-body-final-eq eval-body'·eval-body-eq by simp
also have . . . = eval-final
  unfolding eval-final-def eval-work-final-def unwrapC-def
  by simp
finally show ?thesis .
qed

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:

datatype expr = const nat | add expr expr | disj expr expr | fail

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

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

The following standard list functions underpin the monadic infrastructure:

\begin{verbatim}
fixrec lappend :: 'a llist -> 'a llist -> 'a llist where
lappend·lnil·ys = ys
| lappend·(lcons·x·xs)·ys = lcons·x·(lappend·xs·ys)
\end{verbatim}

\begin{verbatim}
fixrec lconcat :: 'a llist llist -> 'a llist where
lconcat·lnil = lnil
| lconcat·(lcons·x·xs) = lappend·x·(lconcat·xs)
\end{verbatim}

\begin{verbatim}
fixrec lmap :: ('a -> 'b) -> 'a llist -> 'b llist where
lmap·f·lnil = lnil
| lmap·f·(lcons·x·xs) = lcons·(f·x)·(lmap·f·xs)
\end{verbatim}

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

\begin{verbatim}
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))
\end{verbatim}

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:

\begin{verbatim}
definition addS :: S -> S -> S where
addS ≡ (Λ x y. bindS·x·(Λ xv. bindS·y·(Λ yv. returnS·(xv + yv))))
definition disjS :: S -> S -> S where
disjS ≡ lappend
definition failS :: S where
failS ≡ lnil
\end{verbatim}

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}
\texttt{evalS-body :: (expr discr \rightarrow nat discr llist) \rightarrow (expr discr \rightarrow nat discr llist)}

\textbf{where}
\texttt{evalS-body \equiv \Lambda r e. case undiscr e of}
\begin{itemize}
  \item \texttt{const n \Rightarrow returnS \cdot (Discr n)}
  \item \texttt{add e1 e2 \Rightarrow addS \cdot (r \cdot (Discr e1)) \cdot (r \cdot (Discr e2))}
  \item \texttt{disj e1 e2 \Rightarrow disjS \cdot (r \cdot (Discr e1)) \cdot (r \cdot (Discr e2))}
  \item \texttt{fail \Rightarrow failS}
\end{itemize}

\textbf{abbreviation}\texttt{ evalS :: expr discr \rightarrow nat discr llist where evalS \equiv fix \cdot evalS-body}

We aim to transform this evaluator into one using double-barrelled continuations; one will serve as a \textit{"success"} context, taking a natural number into \textit{"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 \texttt{nat discr llist}.

\textbf{type-synonym} \texttt{Obs = nat discr llist}

\textbf{type-synonym} \texttt{Failure = Obs}

\textbf{type-synonym} \texttt{Success = nat discr \rightarrow Failure \rightarrow Obs}

\textbf{type-synonym} \texttt{K = Success \rightarrow Failure \rightarrow Obs}

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

The monad over the continuation type \texttt{K} is as follows:

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

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

Our extra constants are defined as follows:

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

\textbf{definition} \texttt{disjK :: K \rightarrow K \rightarrow K where}
\texttt{disjK \equiv (\Lambda g h. \Lambda s f. g \cdot s \cdot (h \cdot s \cdot f))}

\textbf{definition} \texttt{failK :: K where}
\texttt{failK \equiv \Lambda s f. f}

The continuation semantics is again straightforward:

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

abbreviation $\text{evalK} :: \text{expr discr} \to K$
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 $\text{unwrap}$ is the following function that converts a lazy list into an equivalent continuation representation.

fixrec $\text{SK} :: S \to K$
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 $\text{unwrap} :: (\text{expr discr} \to \text{nat discr llist}) \to (\text{expr discr} \to K)$
where
$$\text{unwrap} \equiv \Lambda r e. \text{SK} \cdot (r \cdot e)$$

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

definition $\text{KS} :: K \to S$
where
$$\text{KS} \equiv (\Lambda k. k \cdot \text{lcons} \cdot \text{lnil})$$

definition $\text{wrap} :: (\text{expr discr} \to K) \to (\text{expr discr} \to \text{nat discr llist})$
where
$$\text{wrap} \equiv \Lambda r e. \text{KS} \cdot (r \cdot e)$$

The worker/wrapper condition follows directly from these definitions.

lemma $\text{KS-SK-id}$:
$$\text{KS} \cdot (\text{SK} \cdot xs) = xs$$
by (induct xs) (simp-all add: $\text{KS-def failK-def}$)

lemma $\text{wrap-unwrap-id}$:
$$\text{wrap} \circ \text{unwrap} = \text{ID}$$
unfolding $\text{wrap-def} \text{ unwrap-def}$
by (simp add: $\text{KS-SK-id cfun-eq-iff}$)

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

lemma $\text{cfun-not-below}$:
$$f \cdot x \not\sqsubseteq g \cdot x \Rightarrow f \not\sqsubseteq g$$
by (auto simp: $\text{cfun-below-iff}$)

lemma $\text{unwrap-wrap-not-under-id}$:
unwrap oo wrap $\not\sqsubseteq$ ID

proof –
let $\text{?witness} = \Lambda e. (\Lambda s f. \text{nil} :: K)$
have (unwrap oo wrap)·$\text{?witness}$(Discr fail)·⊥·(lcons·0·lnil)
  $\not\sqsubseteq$ $\text{?witness}$ (Discr fail)·⊥·(lcons·0·lnil)
  by (simp add: failK-def wrap-def unwrap-def KS-def)
hence (unwrap oo wrap)·$\text{?witness}$$\not\sqsubseteq$ ?witness
  by (fastforce intro!: cfun-not-below)
thus $\text{?thesis}$ by (simp add: cfun-not-below)
qed

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

definition eval-work :: expr discr $\rightarrow$ $K$ where
  eval-work $\equiv$ fix·(unwrap oo evalS-body oo wrap)

definition eval-ww :: expr discr $\rightarrow$ nat discr llist where
  eval-ww $\equiv$ wrap·eval-work

lemma evalS = eval-ww
  unfolding eval-ww-def eval-work-def
  using worker-wrpper-id[OF wrap-unwrap-id]
  by simp

We now show how the monadic operations correspond by showing that 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.

lemma SK-returnS-returnK:
  SK·(returnS·x) = returnK·x
  by (simp add: returnS-def returnK-def failK-def)

lemma SK-lappend-distrib:
  SK·(lappend·xs·ys)·s·f = SK·xs·s·(SK·ys·s·f)
  by (induct xs) (simp-all add: failK-def)

lemma SK-bindS-bindK:
  SK·(bindS·x·g) = bindK·(SK·x)·(SK oo g)
  by (induct x)
    (simp-all add: cfun-eq-iff
     bindS-def bindK-def failK-def
     SK-lappend-distrib)

lemma SK-addS-distrib:
  SK·(addS·x·y) = addK·(SK·x)·(SK·y)
  by (clarsimp simp: ccompose1
    addS-def addK-def failK-def
    SK-bindS-bindK SK-returnS-returnK)

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

by \( \text{simp add: cfun-eq-iff disjS-def disjK-def SK-lappend-distrib} \)

**Lemma SK-failS-failK:**

\[ SK \cdot \text{failS} = \text{failK} \]

**Unfolding failS-def by simp**

These lemmas directly establish the precondition for our all-in-one worker/wrapper and fusion rule:

**Lemma evalS-body-evalK-body:**

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

**Proof:**

\begin{itemize}
  \item **Fix** \( r \cdot e' \cdot s \cdot f \)
  \item **Obtain** \( e :: \text{expr} \)
    where \( ee' = \text{Discr e} \) by \( \text{cases e} \)
  \item **Have**
    \( \text{(unwrap oo evalS-body oo wrap)} \cdot r \cdot (\text{Discr e}) \cdot s \cdot f = \text{(evalK-body oo unwrap oo wrap)} \cdot r \cdot (\text{Discr e}) \cdot s \cdot f \)
  \item **By** \( \text{cases e} \)
    \( \text{(simp-all add: evalS-body-def evalK-body-def unwrap-def SK-returnS-returnK SK-adds-distrib SK-disjS-disjK SK-failS-failK)} \)
  \item **With** \( ee' \)
    **Show**
    \( \text{(unwrap oo evalS-body oo wrap)} \cdot r \cdot e' \cdot s \cdot f = \text{(evalK-body oo unwrap oo wrap)} \cdot r \cdot e' \cdot s \cdot f \)
  \item **By** \( \text{simp} \)
\end{itemize}

**Theorem evalS-evalK:**

\[ \text{evalS} = \text{wrap-evalK} \]

**Using** worker-wrapper-fusion-new[OF \text{wrap-unwrap-id unwrap-strict}]

\[ \text{evalS-body-evalK-body} \]

**By** \( \text{simp} \)

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 \( O(n^2) \) \( \text{nub} \) into an \( O(n \log n) \) one

Andy Gill’s solution, mechanised.
10.1 The nub function.

\texttt{fixrec \text{nub} :: Nat list \rightarrow Nat list}

\texttt{where}

\texttt{nub-nil = inil}

| \texttt{nub}(x :@ xs) = x :@ \text{nub}\cdot(\text{filter}(\Lambda y. x =_{B} y))::xs)

\text{lemma nub-strict[simp]: nub-\bot = \bot}

\text{by fixrec-simp}

\texttt{fixrec \text{nub-body} :: (Nat list \rightarrow Nat list) \rightarrow Nat list \rightarrow Nat list}

\texttt{where}

\texttt{nub-body-f-nil = inil}

| \texttt{nub-body-f}(x :@ xs) = x :@ f\cdot(\text{filter}(\Lambda y. x =_{B} y))::xs)

\text{lemma nub-nub-body-eq: nub = fix\cdot nub-body}

\text{by (rule cfun-eqI, subst nub-def, subst nub-body.unfold, simp)}

10.2 Optimised data type.

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

\texttt{type-synonym NatSet = Nat list}

\texttt{definition SetEmpty :: NatSet where}

\texttt{SetEmpty \equiv inil}

\texttt{definition SetInsert :: Nat \rightarrow NatSet \rightarrow NatSet where}

\texttt{SetInsert \equiv \text{icons}}

\texttt{definition SetMem :: Nat \rightarrow NatSet \rightarrow \text{tr} where}

\texttt{SetMem \equiv \text{bmember}(\text{bpred}(\text{op} =))}

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

\text{by (simp add: SetMem-def)}

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

\text{by (simp add: SetMem-def SetEmpty-def)}

\text{lemma SetMem-SetInsert: SetMem\cdot v\cdot (\text{SetInsert}\cdot x\cdot s) = (SetMem\cdot v\cdot s \text{ orelse} x =_{B} v)}

\text{by (simp add: SetMem-def SetInsert-def)}

AndyG’s new type.

\texttt{domain R = R (lazy resultR :: Nat list) (lazy exceptR :: NatSet)}

\texttt{definition nextR :: R \rightarrow (Nat \ast R) Maybe where}

\texttt{nextR = (\Lambda r. \text{case ldropWhile}(\Lambda x. \text{SetMem}\cdot x\cdot (\text{exceptR}\cdot r))\cdot (\text{resultR}\cdot r)\cdot r)}
\( \text{lnil} \Rightarrow \text{Nothing} \\
| x :: \text{xs} \Rightarrow \text{Just}(x, R \cdot \text{xs} \cdot (\text{exceptR} \cdot r)) \)

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

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

**lemma** \( \text{nextR-lnil}[\text{simp}]: \text{nextR} \cdot (R \cdot \text{lnil} \cdot S) = \text{Nothing} \) by (simp add: nextR-def)

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

**definition**
\[
\text{c2a} :: \text{Nat llist} \rightarrow R \\
\text{c2a} \equiv \Lambda xs. R \cdot xs \cdot \text{SetEmpty}
\]

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

**lemma** \( \text{a2c-strict}[\text{simp}]: \text{a2c} \cdot \bot = \bot \) unfolding a2c-def by simp

**lemma** \( \text{a2c-c2a-id}: \text{a2c} oo \text{c2a} = \text{ID} \)
by (rule cfun-eqI, simp add: a2c-def c2a-def lfilter-const-true)

**definition**
\[
\text{wrap} :: (R \rightarrow \text{Nat llist}) \rightarrow \text{Nat llist} \rightarrow \text{Nat llist} \\
\text{wrap} \equiv \Lambda f xs. f \cdot (\text{c2a} \cdot xs)
\]

**definition**
\[
\text{unwrap} :: (\text{Nat llist} \rightarrow \text{Nat llist}) \rightarrow R \rightarrow \text{Nat llist} \\
\text{unwrap} \equiv \Lambda f r. f \cdot (\text{a2c} \cdot r)
\]

**lemma** \( \text{unwrap-strict}[\text{simp}]: \text{unwrap} \cdot \bot = \bot \) unfolding unwrap-def by (rule cfun-eqI, simp)

**lemma** \( \text{wrap-unwrap-id}: \text{wrap} oo \text{unwrap} = \text{ID} \)
using cfun-fun-cong[OF a2c-c2a-id]
by \(((\text{rule cfun-eqI}), \text{simp} \text{ add}: \text{unwrap-def wrap-def})\)

Equivalences needed for later.

**lemma** \( \text{TR-deMorgan}: \text{neg} \cdot (x \ orelse y) = (\text{neg} \cdot x \ \text{andalso} \ \text{neg} \cdot y) \)
by (rule trE[where \( p=x \)], simp-all)

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

36
apply (cases L, simp-all)
apply (case-tac h-a-llist)
apply simp
done

lemma case-a2c-case-caseR:
  (case a2c-w of lnil ⇒ f | x :@ xs ⇒ g-x-x)
= (case nextR-w of Nothing ⇒ f | Just-(x, r) ⇒ g-x-(a2c-r)) (is ?lhs = ?rhs)
proof –
have ?rhs = (case (case ldropWhile-(Λ x. SetMem-x-(exceptR-w)))-resultR-w of
lnil ⇒ Nothing
  | x :@ xs ⇒ Just-(x, R-x-(exceptR-w))) of Nothing ⇒ f | Just-(x, r) ⇒ g-x-(a2c-r))
  by (simp add: nextR-def)
also have . . . = (case ldropWhile-(Λ x. SetMem-x-(exceptR-w)))-resultR-w of
lnil ⇒ f | x :@ xs ⇒ g-x-(a2c-(R-x-(exceptR-w))))
using case-maybe-case[where L=ldropWhile-(Λ x. SetMem-x-(exceptR-w)))-resultR-w
  and f=f and g=Λ x r. g-x-(a2c-r) and h=Λ x xs. (x, R-x-(exceptR-w))]
  by simp
also have . . . = ?lhs
  apply (simp add: a2c-def)
  apply (cases resultR-w)
  apply simp-all
  apply (rule-tac p=SetMem-a-(exceptR-w) in trE)
  apply simp-all
  apply (induct-tac list)
  apply simp-all
  apply (rule-tac p=SetMem-aa-(exceptR-w) in trE)
  apply simp-all
  done
finally show ?lhs = ?rhs by simp
qed

lemma filter-filterR: lfilter-(neg oo (Λ y. x =B y))-a2c-r = a2c-(filterR-x-r)
using filter-filter[where p=Tr.neg oo (Λ y. x =B y) and q=Λ v. Tr.neg-(SetMem-v-(exceptR-r))]
unfolding a2c-def filterR-def
by (cases r, simp-all add: SetMem-SetInsert TR-deMorgan)

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

definition
nub-body' :: (R → Nat list) → R → Nat list where
nub-body' ≡ Λ f v. case a2c-r of lnil ⇒ lnil
  | x :@ xs ⇒ x :@ f-(c2a-(lfilter-(neg oo (Λ y. x =B y))))-xs)

lemma nub-body-nub-body'-eq: unwrap oo nub-body oo wrap = nub-body'
unfolding nub-body-def nub-body'-def unwrap-def wrap-def a2c-def c2a-def
by ((rule cfun-eqI)+
    , case-tac lfilter(Λ v. Tr.neg(ΣMem-v.(exceptR·r))))·(resultR·r)
    , simp-all add: fix-const)

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

lemma nub-body'-nub-body''-eq: nub-body' = nub-body''
proof (rule cfun-eqI)+
  fix f r show nub-body'·f·r = nub-body''·f·r
  unfolding nub-body'-def nub-body''-def
  using case-a2c-case-caseR[where f=Nil and g=Λ x xs, x :@ f · (lfilter·(Tr.neg
       oo (Λ y. x =B y))·xs)] and w=r]
  by simp
qed

definition
  nub-body'''' :: (R → Nat list) → R → Nat list 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)
  unfolding nub-body''-def nub-body''''-def wrap-def unwrap-def
by ((rule cfun-eqI)+, simp add: filter-filterR)

Finally glue it all together.

lemma nub-wrap-nub-body''''': nub = wrap·(fix·nub-body''''')
using worker-wrapper-fusion-new[OF wrap-unwrap-id unwrap-strict, where body=nub-body]
  nub-nub-body-eq
  nub-body-nub-body'-eq
  nub-body'-nub-body''-eq
  nub-body''-nub-body''''-eq
by simp

end

11 Optimise “last”.

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

11.1 The last function.

fixrec llast :: 'a list → 'a
  where
    llast·(x :@ yys) = (case yys of lnil ⇒ x | y :@ ys ⇒ llast·yys)
lemma \textit{llast-strict} \{simp\}: \textit{llast} \cdot \bot = \bot \\
by \textit{fixrec-simp}

\textbf{fixrec} \textit{llast-body} :: \('a \textit{l}\textit{list} \rightarrow 'a) \rightarrow \textit{'}a \textit{l}\textit{list} \rightarrow 'a \\
\textbf{where} \\
\textit{llast-body}.f.(x :@ yys) = (\textit{case yys of} \textit{nil} \Rightarrow x \mid y :@ yys \Rightarrow f \cdot yys)

lemma \textit{llast-llast-body} by (\textit{rule cfun-eqI}, \textit{subst llast-def}, \textit{subst llast-body.unfold}, \textit{simp})

definition \textit{wrap} :: ('a \rightarrow 'a \textit{l}\textit{list} \rightarrow 'a) \rightarrow ('a \textit{l}\textit{list} \rightarrow 'a) where \\
\textit{wrap} \equiv \Lambda f \cdot (x :@ xs). f \cdot x \cdot xs \\
definition \textit{unwrap} :: ('a \textit{l}\textit{list} \rightarrow 'a) \rightarrow ('a \rightarrow 'a \textit{l}\textit{list} \rightarrow 'a) where \\
\textit{unwrap} \equiv \Lambda f x xs \cdot f \cdot (x :@ xs)

lemma \textit{unwrap-strict} \{simp\}: \textit{unwrap} \cdot \bot = \bot \\
\textbf{unfolding} \textit{unwrap-def} by ((\textit{rule cfun-eqI})+, \textit{simp})

lemma \textit{wrap-unwrap-ID}: \textit{wrap} oo \textit{unwrap} oo \textit{llast-body} = \textit{llast-body} \\
\textbf{unfolding} \textit{llast-body-def} \textit{wrap-def} \textit{unwrap-def} \\
\textbf{apply} (\textit{rule cfun-eqI})+ \\
\textbf{apply} (\textit{case-tac} xa) \\
\textbf{apply} (\textit{simp-all add: fix-const}) \\
\textbf{done}

definition \textit{llast-worker} :: ('a \rightarrow 'a \textit{l}\textit{list} \rightarrow 'a) \rightarrow 'a \rightarrow 'a \textit{l}\textit{list} \rightarrow 'a \textit{where} \\
\textit{llast-worker} \equiv \Lambda r x yys \cdot \textit{case yys of} \textit{nil} \Rightarrow x \mid y :@ yys \Rightarrow r \cdot y \cdot yys \\
definition \textit{llast}': 'a \textit{l}\textit{list} \rightarrow 'a \textit{where} \\
\textit{llast}' \equiv \textit{wrap} \cdot (\textit{fix}-\textit{llast-worker})

lemma \textit{llast-worker-llast-body}: \textit{llast-worker} = \textit{unwrap} oo \textit{llast-body} oo \textit{wrap} \\
\textbf{unfolding} \textit{llast-body-def} \textit{llast-body-def} \textit{wrap-def} \textit{unwrap-def} \\
\textbf{apply} (\textit{rule cfun-eqI})+ \\
\textbf{apply} (\textit{case-tac} xb) \\
\textbf{apply} (\textit{simp-all add: fix-const}) \\
\textbf{done}

lemma \textit{llast}'-llast: \textit{llast}' = \textit{llast} (\textit{is} \ ?lhs = \ ?rhs) \\
\textbf{proof} \\
\textbf{have} \ ?rhs = \textit{fix}-\textit{llast-body} by (\textit{simp only: llast-llast-body}) \\
\textbf{also have} \ldots = \textit{wrap} \cdot (\textit{fix} \cdot (\textit{unwrap} oo \textit{llast-body} oo \textit{wrap})) \\
\hspace{1em} by (\textit{simp only: worker-wraper-body} (\textit{OF} \textit{wrap-unwrap-ID})) \\
\textbf{also have} \ldots = \textit{wrap} \cdot (\textit{fix} \cdot \textit{llast-worker}) \\
\hspace{1em} by (\textit{simp only: llast-worker-llast-body}) \\
\textbf{also have} \ldots = \ ?lhs \textit{unfolding} \textit{llast}'-\textit{llast-def} by \textit{simp} \\
\textbf{finally show} \ ?thesis by \textit{simp}
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 \( f :: \text{Nat} \rightarrow a \) 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


