Mechanising the worker/wrapper transformation

Peter Gammie

August 16, 2018

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 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
1 Introduction

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

2 Fixed-point theorems for program transformation

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

2.1 The rolling rule

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

lemma rolling-rule-ltr: fix·(g oo f) ⊑ g·(fix·(f oo g))
proof –
  have g·(fix·(f oo g)) ⊑ g·(fix·(f oo g))
  by (rule below-refl) — reflexivity
  hence g·((f oo g)·(fix·(f oo g))) ⊑ g·(fix·(f oo g))
  using fix-eq[where F=f oo g] by simp — computation
  hence (g oo f)·(g·(fix·(f oo g))) ⊑ g·(fix·(f oo g))
  by simp — re-associate (oo)
  thus fix·(g oo f) ⊑ g·(fix·(f oo g))
using \texttt{fix-least-below} by \texttt{blast} — induction

\texttt{qed}

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

\textbf{proof} —

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

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

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

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

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

\texttt{qed}

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

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

\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}[scale=0.5]
  \node (A) at (0,0) {$f$};
  \node (B) at (2,0) {$g$};
  \node (C) at (2,2) {$h$};
  \node (D) at (0,2) {$\text{fix } f$};
  \node (E) at (4,2) {$\text{fix } h$};

  \draw[->] (A) -- (B);
  \draw[->] (B) -- (C);
  \draw[->] (C) -- (E);
  \draw[->] (D) -- (B);
\end{tikzpicture}
\end{center}

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 \texttt{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{equation}
\begin{align*}
\text{adm} \ (\lambda x. \ ?P \ (\text{fst} \ x) \ (\text{snd} \ x)) \\
?P \perp \perp \quad \bigwedge x y. \ ?P \ x y \\
\quad \quad \quad \ ?P \ (\text{fix} \cdot ?F) \ (\text{fix} \cdot ?G)
\end{align*}
\end{equation}

in a very straightforward way:

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

\begin{itemize}
\item \texttt{assumes} \( g \perp = \perp \)
\item \texttt{assumes} \( g \circ f = h \circ g \)
\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 \( \text{comp} = \text{fix} \text{body} \) for some \( \text{body} :: A \rightarrow A \) and a pair of functions \( \text{wrap} :: B \rightarrow A \) and \( \text{unwrap} :: A \rightarrow B \) where \( \text{wrap} \circ \text{unwrap} = \text{id}_A \), we have:

\[
\text{comp} = \text{wrap} \text{ work}
\text{ work} :: B
\text{ work} = \text{fix} (\text{unwrap} \circ \text{body} \circ \text{wrap})
\]

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

\begin{verbatim}
lemma worker-wrapper-id:
  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{comp-body}: \text{computation} = \text{fix-body}
  shows \text{computation} = \text{wrap}·\text{fix}·(\text{unwrap} \circ \text{body} \circ \text{wrap})
proof –
  from \text{comp-body} have \text{computation} = \text{fix}·\text{id} oo \text{body}
\end{verbatim}
... = 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 \( \text{work} \): \( \text{work} = \text{fix}(\text{unwrap} \circ \text{body} \circ \text{wrap}) \)

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

**Proof**

- have \( (\text{unwrap} \circ \text{wrap}) \cdot \text{work} = (\text{unwrap} \circ \text{wrap}) \cdot (\text{fix}(\text{unwrap} \circ \text{body} \circ \text{wrap})) \)
  
  using \( \text{work} \) by simp

- also have \( \ldots = (\text{unwrap} \circ \text{wrap}) \cdot (\text{fix}(\text{unwrap} \circ \text{body} \circ \text{wrap} \circ \text{unwrap} \circ \text{wrap})) \)
  
  using \( \text{wrap-unwrap} \) by (simp add: assoc-oo)

- also have \( \ldots = \text{fix}(\text{unwrap} \circ \text{body} \circ \text{wrap}) \)
  
  using \( \text{rolling-rule}[\text{where } f = \text{unwrap} \circ \text{body} \circ \text{wrap} \text{ and } g = \text{unwrap} \circ \text{wrap}] \)
  
  by (simp add: assoc-oo)

- also have \( \ldots = \text{fix}(\text{unwrap} \circ \text{body} \circ \text{wrap}) \)
  
  using \( \text{wrap-unwrap} \) by (simp add: assoc-oo)

- finally show \( \text{thesis} \) using \( \text{work} \) 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 \( f = \text{body} \) yields a rule of the form \( \text{body} \Rightarrow f \), one can always derive \( f = f \). Clearly this has dire implications for the preservation of termination behaviour.

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

\[
\frac{f \cdot \mathcal{A} = \mathcal{A}}{\text{fix}\cdot f \subseteq \mathcal{A}} \quad \text{fix}\_\text{least}
\]
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** wrap-unwrap: \( \text{wrap} \circ \text{unwrap} = \text{ID} \)
- **Assumes** work: \( \text{work} = \text{fix} \cdot (\text{unwrap} \circ \text{body} \circ \text{wrap}) \)
- **Assumes** 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** work by (simp add: fix-eq[symmetric])
  - **Also have** \( \ldots = (\text{body}' \circ \text{unwrap} \circ \text{wrap}) \cdot \text{work} \)
    - **Using** body' by simp
    - **Also have** \( \ldots = (\text{unwrap} \circ \text{body} \circ \text{wrap}) \cdot \text{work} \)
      - **Using** work by (simp add: fix-eq[symmetric])
      - **Also have** \( \ldots = (\text{body}' \circ \text{unwrap} \circ \text{wrap} \circ \text{unwrap} \circ \text{body} \circ \text{wrap}) \cdot \text{work} \)
        - **Using** simp
        - **Also have** \( \ldots = (\text{unwrap} \circ \text{body} \circ \text{wrap}) \cdot \text{work} \)
          - **Using** unwrap-unwrap by (simp add: assoc-oo)
          - **Also have** \( \ldots = \text{body}' \cdot \text{work} \)
            - **Using** 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** \( A = A \)
domain \( B = B \) (lazy \( A \))

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

\[
\begin{align*}
\text{fixrec} \ \text{wrap} &: B \to A \\
\text{where} \ \text{wrap} \cdot (B \cdot a) &= a
\end{align*}
\]

\[
\begin{align*}
\text{fixrec} \ \text{unwrap} &: A \to B \\
\text{where} \ \text{unwrap} &= B
\end{align*}
\]

Discharging the worker/wrapper hypothesis is similarly routine.

\[
\text{lemma} \ \text{wrap-unwrap}: \ \text{wrap} \circ \text{unwrap} = \text{Id}
\]

by (\( \text{simp add: cfun-eq-iff} \))

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

\[
\begin{align*}
\text{fixrec} \ \text{body} &: A \to A \\
\text{where} \ \text{body} \cdot r &= A
\end{align*}
\]

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

\[
\begin{align*}
\text{fixrec} \ \text{body}' &: B \to B \\
\text{where} \ \text{body}' \cdot (B \cdot a) &= B \cdot A
\end{align*}
\]

As explained above, we set up the fusion opportunity:

\[
\text{lemma} \ \text{body-body'}: \ \text{unwrap} \circ \text{body} \circ \text{unwrap} = \text{body'} \circ \text{unwrap} \circ \text{unwrap} \circ \text{wrap}
\]

by (\( \text{simp add: cfun-eq-iff} \))

This result depends crucially on \( \text{unwrap} \) being non-strict.

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

\[
\text{lemma} \ \text{fix-body'} \subseteq \text{fix} \cdot (\text{unwrap} \circ \text{body} \circ \text{unwrap})
\]

by (\( \text{rule fusion-partially-correct[OF \text{wrap-unwrap refl body-body'}] \))

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

\[
\text{lemma} \ \neg \ \text{fix} \cdot (\text{unwrap} \circ \text{body} \circ \text{unwrap}) \subseteq \text{fix-body'}
\]

proof –

\[
\begin{align*}
\text{have} \ l: \ &\text{fix} \cdot (\text{unwrap} \circ \text{body} \circ \text{unwrap}) = B \cdot A \\
&\text{by (subst \text{fix-eq} \ simp)} \\
\text{have} \ r: \ &\text{fix-body'} = \bot \\
&\text{by (simp add: \text{fix-strict})} \\
\text{from} \ l \ r \ \text{show} \ \neg \text{thesis} \ \text{by simp}
\end{align*}
\]

qed

This trick works whenever \( \text{unwrap} \) is not strict. In the following section we show that requiring \( \text{unwrap} \) 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 \( \text{wrap} \circ \text{strictify-unwrap} = \text{ID} \)

**proof** (rule cfun-eqI)

fix \( x \)

from assms

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

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 \( \text{worker}-\text{wrapper-id} \). 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 \( \text{worker}-\text{wrapper-id} \) and the ubiquitous \( \text{lfp-fusion} \) rule.

**lemma** \( \text{worker}-\text{wrapper-fusion-new} \):

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

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

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

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

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

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

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

**proof**

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

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

by (simp add: assoc-oo)

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

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 \( \text{unwrap-unwrap} \) 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 \texttt{parallel\_fix\_ind}:

\textbf{lemma} worker-wrapper-fusion-new-general:

\texttt{fixes wrap :: 'b::pcpo \rightarrow 'a::pcpo}
\texttt{fixes unwrap :: 'a \rightarrow 'b}
\texttt{assumes wrap-unwrap: wrap oo unwrap = (ID :: 'a \rightarrow 'a)}
\texttt{assumes unwrap-strict: unwrap\cdot\bot = \bot}
\texttt{assumes body-body': \forall r. (unwrap oo wrap)\cdot r = r}
\texttt{\implies (unwrap oo body oo wrap)\cdot r = body'\cdot r}
\texttt{shows fix\cdot body = wrap\cdot (fix\cdot body')}

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

\textbf{type-synonym} ‘a H = ‘a list \rightarrow ‘a list

\textbf{definition}
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 \( \text{unwrap} \perp = \perp \), define:

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

(the worker/wrapper transformation)

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

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

(worker/wrapper fusion)

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

\[
\begin{align*}
\text{list2H} :: \forall a \cdot \text{lalist} &\to \forall a \cdot \text{H} \\
\text{list2H} &\equiv \text{lappend}
\end{align*}
\]

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

by \( \text{rule cfun-eqI, simp add: list2H-def} \)

**definition**

\[
\text{H2list} :: \forall a \cdot \text{H} \to \forall a \cdot \text{lalist}
\]

\[
\text{H2list} \equiv \Lambda f. f \cdot \text{lnil}
\]

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

**proof**(rule cfun-eqI)

fix \( \text{zs} \)

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

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

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

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

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

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

qed

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

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

by \( \text{rule cfun-eqI, simp add: H2list-def list2H-def} \)

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

**fixrec** \( \text{lrev} :: \forall a \cdot \text{lalist} \to \forall a \cdot \text{lalist} \)

11
where
\[
l\text{rev-nil} = \text{nil} \\
l\text{rev}(x :@ xs) = \text{rev} \cdot xs :++ (x :@ \text{nil})
\]

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

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

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

where
\[
l\text{rev-body-r-nil} = \text{nil} \\
l\text{rev-body-r}(x :@ xs) = r \cdot xs :++ (x :@ \text{nil})
\]

**lemma** \text{lrev-body-strict}[simp]: \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** (\text{rule cfun-eqI}, \text{subst \text{lrev-def}}, \text{subst \text{lrev-body}}, \text{unfold}, \text{simp})

Wrap / unwrap functions.

**definition**
\[
\text{unwrapH} :: ('a \text{llist} \rightarrow 'a \text{llist}) \rightarrow 'a \text{H}
\text{unwrapH} \equiv \Lambda f \cdot \text{list2H} \cdot \text{f} \cdot \text{xs}
\]

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

**definition**
\[
\text{wrapH} :: ('a \text{llist} \rightarrow 'a \text{H}) \rightarrow 'a \text{llist}
\text{wrapH} \equiv \Lambda f \cdot \text{H}2\text{list} \cdot \text{f} \cdot \text{xs}
\]

**lemma** \text{wrapH-unwrapH-id}: \text{wrapH} \circ \text{unwrapH} = \text{ID} (\text{is \text{lhs} = \text{rhs}})
**proof** (rule cfun-eqI)+
\text{fix} f \text{ xs}
\text{have} ?\text{lhs} \cdot \text{f} \cdot \text{xs} = \text{H}2\text{list} \cdot \text{list2H} \cdot \text{f} \cdot \text{xs} \text{ by simp add: rewrite-def rewrite-def}
\text{also have} \ldots = \text{H}2\text{list} \circ \text{list2H} \cdot \text{f} \cdot \text{xs} \text{ by simp}
\text{also have} \ldots = \text{ID} \cdot \text{f} \cdot \text{xs} \text{ by simp only: \text{H}2\text{list-list2H-inv}}
\text{also have} \ldots = ?\text{rhs} \cdot \text{f} \cdot \text{xs} \text{ by simp}
\text{finally show} ?\text{lhs} \cdot \text{f} \cdot \text{xs} = ?\text{rhs} \cdot \text{f} \cdot \text{xs} .
\text{qed}

5.2 Gill/Hutton-style worker/wrapper.

**definition**
\[
\text{lrev-work} :: 'a \text{llist} \rightarrow 'a \text{H}
\text{lrev-work} \equiv \text{fix} \cdot (\text{unwrapH} \circ \text{lrev-body} \circ \text{wrapH})
\]
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 lrev-body1)
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
lemma lrev-body2-lrev-body1-eq: lrev-body2 = lrev-body1
  by ((rule cfun-eqI)+
     , (subst lrev-body1.unfold, subst lrev-body2.unfold)

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

Simplify.

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

lemma lrev-body3-strict[simp]: lrev-body3·r·⊥ = ⊥
  by fixrec-simp

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 -
{  
  have list2H oo wrapH·lrev-work2 = unwrapH·(wrapH·lrev-work2)
    by (rule cfun-eqI, simp add: unwrapH-def)
  also have ... = (unwrapH oo wrapH)·lrev-work2 by simp
  also have ... = lrev-work2
    apply ...
    apply (rule worker-wrapper-fusion[OF wrapH-unwrapH-id, where body=lrev-body])
    apply (auto iff: lrev-body2-lrev-body1-eq lrev-body-lrev-body1-eq lrev-work2-def lrev-work1-def)
  finally have list2H oo wrapH·lrev-work2 = lrev-work2 .
}
  thus ?thesis using cfun-eq-iff[where f=list2H oo wrapH·lrev-work2 and g=lrev-work2]
    by auto

qed

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

lemma lrev-work3 ⊑ lrev-work2
unfolding lrev-work3-def
proof(rule fix-least)
{
  fix xs have lrev-body3·lrev-work2·xs = lrev-work2·xs
  proof(cases xs)
    case bottom thus ?thesis by simp

14
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 \texttt{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.

\begin{verbatim}
lemma lrev-work3-lrev-work2-eq: lrev-work3 = lrev-work2 (is \?lhs = \?rhs)
proof(rule cfun-eqI)
fix x
show \?lhs·x = \?rhs·x
proof(induct x)
  show lrev-work3·⊥ = lrev-work2·⊥.
    apply (unfold lrev-work3-def lrev-work2-def)
    apply (subst fix-eq[where \(F=lrev-body2\)])
    apply (subst fix-eq[where \(F=lrev-body3\)])
    by (simp add: lrev-body3.unfold lrev-body2.unfold)
next
  show lrev-work3·nil = lrev-work2·nil
    apply (unfold lrev-work3-def lrev-work2-def)
    apply (subst fix-eq[where \(F=lrev-body2\)])
    apply (subst fix-eq[where \(F=lrev-body3\)])
    by (simp add: lrev-body3.unfold lrev-body2.unfold)
next
  fix a l assume lrev-work3·l = lrev-work2·l
thus lrev-work3·(a :: l) = lrev-work2·(a :: l)
  apply (unfold lrev-work3-def lrev-work2-def)
  apply (subst fix-eq[where \(F=lrev-body2\)])
  apply (subst fix-eq[where \(F=lrev-body3\)])
end
\end{verbatim}
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 (subt lrev-body2.unfold, subt 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 ... = 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-body-final ≡ fix·lrev-body-final

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

lemma lrev-body-final-lrev-body3-eq': lrev-body-final·r·xs = lrev-body3·r·xs
apply (subt lrev-body-final.unfold)
apply (subt 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-work1-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 ... = 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·⊥ = ⊥
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 unfacB :: (Nat → Nat) → UNat → UNat⊥ where
unfacB ≡ Λ f. unbox oo f oo box

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

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

lemma wrapB-unfacB-body:
  assumes strictF: f·⊥ = ⊥
  shows (wrapB oo unfacB)·f = f (is ?lhs = ?rhs)
proof (rule cfun-eqI)
  fix x :: Nat
  have ?lhs·x = unbox·x ≫= (Λ x'. unfacB·f·x' ≫= box)
    unfolding wrapB-def by simp
  also have ... = unbox·x ≫= (Λ x'. unbox·(f·(box·x'))) ≫= box
    unfolding unfacB-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 (unfacB 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 unfacB oo fac-body = fac-body
    using wrapB-unfacB-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 unfac=unfacB]
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 =₀ 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)]
    bbind-case-distr-strict[where f=Λ y. box·x * y and h=box]
  unfolding fac-body'-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 = fac-wrap by (rule fac-fac-ww-eq)
  also have ... = wrapB-fac-work by (simp only: fac-wrap-def)
  also have ... = wrapB-fix-unwrapB oo fac-body oo wrapB) by (simp only: fac-wrap-def)
  also have ... = wrapB-fix-fac-body by (simp only: fac-body-fac-body)
  also have ... = wrapB-fix-fac-work-final by (simp only: fac-body-final-fac-body)
  also have ... = wrapB-fix-fac-body by (simp only: fac-body-final-fac-body)
  also have ... = fac-final by (simp add: fac-final-def wrapB-def)
  finally show ?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 (>>, in the homomorphism.

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

type-synonym UNatAcc = UNat → UNat⊥
definition n2a :: UNat → UNatAcc where
  n2a ≡ Λ m n. up·(m ♭ n)
definition a2n :: UNatAcc → UNat⊥ where
  a2n ≡ Λ a. a·1

lemma a2n-strict[simp]: a2n·⊥ = ⊥
  unfolding a2n-def by simp

lemma a2n-n2a: a2n·(n2a·u) = up·u
  unfolding a2n-def n2a-def by (simp add: uMult-arithmetic)

lemma A-hom-mult: n2a·(x ♭ y) = (n2a·x) >>= n2a·y)
  unfolding n2a-def bKleisli-def by (simp add: uMult-arithmetic)
definition unwrapA :: (UNat → UNat⊥) → UNat → UNatAcc where
  unwrapA ≡ Λ f n. f·n >>= n2a

lemma unwrapA-strict[simp]: unwrapA·⊥ = ⊥
  unfolding unwrapA-def by (rule cfun-eqI) simp
definition
wrapA :: (UNat → UNatAcc) → UNat → UNat⊥ where
wrapA ≡ Λ f. a2n oo f

lemma wrapA-unwrapA-id: wrapA oo unwrapA = ID
unfolding wrapA-def unwrapA-def
apply (rule cfun-eqI)+
apply (case-tac x·xa)
apply (simp-all add: a2n-n2a)
done

Some steps along the way.

definition
fac-acc-body1 :: (UNat → UNatAcc) → UNat → UNatAcc where
fac-acc-body1 ≡ Λ r n.
  if n = 0 then n2a·1 else wrapA·r·(n − # 1) >>= (Λ res. n2a·(n *# res))

lemma fac-acc-body1-fac-body-final-eq: fac-acc-body1 = unwrapA oo fac-body-final oo wrapA
unfolding fac-acc-body1-def fac-body-final-def wrapA-def unwrapA-def
by (rule cfun-eqI)+ simp

Use the homomorphism.

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

lemma fac-acc-body2-body1-eq: fac-acc-body2 = fac-acc-body1
unfolding fac-acc-body1-def fac-acc-body2-def
by (rule cfun-eqI)+ (simp add: A-hom-mult)

Apply worker/wrapper.

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

lemma fac-acc-body3-body2: fac-acc-body3 oo (unwrapA oo wrapA) = fac-acc-body2 (is ?lhs=?rhs)
proof (rule cfun-eqI)+
fix r n acc
show ((fac-acc-body3 oo (unwrapA oo wrapA))·r·n·acc) = fac-acc-body2·r·n·acc
  unfolding fac-acc-body2-def fac-acc-body3-def unwrapA-def
  using bbind-case-distr-strict [where f=Λ y. n2a·n >>= y and h=n2a, symmetric]
  by 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: cfcomp1)
  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
\[ i-th \cdot (x \&\& xs) = \text{Nat-case} \cdot (i-th \cdot xs) \]

**abbreviation**

\[
i-th-syn :: \text{'}a \text{ Stream } \Rightarrow \text{'}a \text{ (infixl !! 100) where } s \&\& i \equiv i-th \cdot s \cdot i
\]

The infinite stream of natural numbers.

**fixrec** nats :: Nat Stream

**where**

\[
nats = 0 \&\& \text{smap} \cdot (\Lambda x. 1 + x) \cdot nats
\]

### 7.2 The wrapper/unwrapper functions.

**definition**

\[
\text{unwrapS'} :: (\text{Nat } \Rightarrow \text{'}a) \Rightarrow \text{'}a \text{ Stream where }
\text{unwrapS'} \equiv \Lambda f . \text{smap} \cdot f \cdot nats
\]

**lemma** unwrapS'-unfold: unwrapS' \( f = f \cdot 0 \&\& \text{smap} \cdot (f oo (\Lambda x. 1 + x)) \cdot nats \)

**fixrec** unwrapS :: (Nat \( \Rightarrow \) 'a) \( \Rightarrow \) 'a Stream

**where**

\[
\text{unwrapS} \cdot f = f \cdot 0 \&\& \text{unwrapS} \cdot (f oo (\Lambda x. 1 + x))
\]

The two versions of \text{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:

\[
\text{unwrapS} = \text{unwrapS'} \text{ is } \text{lhs} = \text{rhs}
\]

**proof** (**rule cfun-eqI**)

fix \( f \) show \( ?\text{lhs} \cdot f = ?\text{rhs} \cdot f \)

**proof** (**coinduct rule: Stream.coinduct**)

let \( ?R = \lambda s s'. (\exists f. s = f \cdot 0 \&\& \text{unwrapS} \cdot (f oo (\Lambda x. 1 + x)) \wedge s' = f \cdot 0 \&\& \text{smap} \cdot (f oo (\Lambda x. 1 + x)) \cdot nats) \)

show Stream-bisim \( ?R \)

**proof**

fix \( s s' \) assume \( ?R \ s s' \)

then obtain \( f \) where \( fs: s = f \cdot 0 \&\& \text{unwrapS} \cdot (f oo (\Lambda x. 1 + x)) \)

and \( fs': s' = f \cdot 0 \&\& \text{smap} \cdot (f oo (\Lambda x. 1 + x)) \cdot nats \)

by blast

have \( ?R (\text{unwrapS} \cdot (f oo (\Lambda x. 1 + x))) \ (\text{smap} \cdot (f oo (\Lambda x. 1 + x)) \cdot nats) \)

by (** rule exI[where x=foo (\Lambda x. 1 + x)]**) subst unwrapS.unfold, subst nats.unfold, simp add: smap-smap)

with \( fs \ fs' \)

show \( (s = \bot \wedge s' = \bot) \)

\[ \exists h t t'. \]

\[ (\exists f. t = f \cdot 0 \&\& \text{unwrapS} \cdot (f oo (\Lambda x. 1 + x)) \wedge t' = f \cdot 0 \&\& \text{smap} \cdot (f oo (\Lambda x. 1 + x)) \cdot nats) \]

\[ \wedge s = h \&\& t \wedge s' = h \&\& t' \) by best

qed
show \( \text{R}(\text{lhs}\cdot f) (\text{rhs}\cdot f) \)

proof -
  have lhs: \( \text{lhs}\cdot f = f\cdot0 \&\& \text{unwrap}\cdot (f\ oo\ (\Lambda\ x.\ 1 + x)) \) by (subst \text{unwrap}.\ unfold, simp)
  have rhs: \( \text{rhs}\cdot f = f\cdot0 \&\& \text{smap}\cdot (f\ oo\ (\Lambda\ x.\ 1 + x))\cdot\text{nats} \) by (rule unwrap\'-\ unfold)
  from lhs rhs show \( \text{thesis} \) by best
  qed
  qed
  qed

definition
wrapS :: 'a Stream \(\rightarrow\) Nat \(\rightarrow\) '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: \( f::\text{Nat}\ \rightarrow\ 'a\cdot\bot = \bot \)
  shows unwrap\cdot (f\ oo\ (\Lambda\ x.\ 1 + x)) \(\equiv\) \( f\cdot\text{nats} \)
  using strictF
proof(induct n arbitrary: \( f \) rule: Nat-induct)
  case bottom with strictF show \( ?\text{case} \) by simp
next
  case zero thus \( ?\text{case} \) by (subst unwrap.\ unfold, simp)
next
  case (Suc \( i\) \( f \))
  have unwrap\cdot (f\ oo\ (\Lambda\ x.\ 1 + x)) \(\equiv\) \( f\cdot\text{nats} \)
    by (subst unwrap.\ unfold, simp)
  also from Suc have \( \ldots = \text{unwrap}\cdot (f\ oo\ (\Lambda\ x.\ 1 + x)) \) \(\equiv\) \( i\ ) by simp
  also from Suc have \( \ldots = (f\ oo\ (\Lambda\ x.\ 1 + x))\cdot i \) by simp
  also have \( \ldots = (\Lambda\ n.\ r\cdot n + r\cdot(n + 1)) \) by (simp add: plus-commute)
  finally show \( ?\text{case} \).
  qed

lemma wrapS-unwrapS-id: \( f\cdot\bot = \bot \Longrightarrow (\text{wrap}\cdot f\ oo\ \text{unwrap})\cdot f = f \)
  by (rule cfun-eqI, simp add: wrapS-unwrapS-id' wrapS-def)

7.3 Fibonacci example.

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

definition
fib :: Nat \(\rightarrow\) Nat where
fib \(\equiv\) fix\cdot fib-body

24
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 Δ fib-work

lemma wrapS-unwrapS-fib-body:
  wrapS oo unwrapS oo fib-body = fib-body
proof
  fix r show
    (wrapS oo unwrapS oo fib-body)·r = fib-body·r
  using wrapS-unwrapS-id[where f = fib-body·r] by simp
qed

lemma fib-ww-eq: fib = fib-wrap
  using worker-wrapper-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 Δ 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

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 = Λ r. Nat-case·1·(Nat-case·1·(Λ n′. r!!n′ + r!!(n′ + 1)))
  let ?mr = Λ (fief :: Nat Stream, fff). (smap·fff·nats, ?wb·fief)
  have ?lhs = fst (fix·?mr)
    by (simp add: fib-work-final-def split-def csplit-def)

1The interested reader can find some historical commentary in Harel (1980); Sangiorgi
(2009).
also have \ldots = (\mu \text{fwf}, \text{fst} (\?mr \cdot (\text{fwf}, \mu \text{fff}. \text{snd} (\?mr \cdot (\text{fwf}, \text{fff}))))))
using \text{fix-cprod}[\text{where } F=\?mr] \text{ by simp}
also have \ldots = (\mu \text{fwf}, \text{smap}(\mu \text{fff}. \?wb \cdot \text{fwf}) \cdot \text{nats}) \text{ by simp}
also have \ldots = (?rhs)
unfolding \text{fib-body-def} \text{ fib-def} \text{ unwrapS-unwrapS'-eq} \text{ unwrapS'-def} \text{ wrapS-def}
by (\text{simp add: cfcomp1})
finally show ?thesis.
qed

lemma \text{fib-final-fib-eq}:
\text{fib-final} = \text{fib} (\text{is } \?lhs = \?rhs)
proof –
have \?lhs = (\Lambda n. \text{fib-work-final} !! n) \text{ by (simp add: fib-final-def)}
also have \ldots = (\Lambda n. \text{fib-work} !! n) \text{ by (simp only: fib-work-final-fib-work-eq)}
also have \ldots = \text{fib-wrap} \text{ by (simp add: fib-wrap-def wrapS-def)}
also have \ldots = \?rhs \text{ by (simp only: fib-wv-eq)}
finally show ?thesis.
qed

8 Tagless interpreter via double-barreled continuations

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

definition \text{val2cont} :: ‘a \rightarrow ‘a \text{Cont} where
\text{val2cont} \equiv (\Lambda a c. \text{c-a})

definition \text{cont2val} :: ‘a \text{Cont} \rightarrow ‘a where
\text{cont2val} \equiv (\Lambda f. f \cdot \text{ID})

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

domain \text{Expr} =
\text{Val} (\text{lazy} \text{ val::Nat})
| \text{Add} (\text{lazy} \text{ addl::Expr}) (\text{lazy} \text{ addr::Expr})
| \text{Throw}
| \text{Catch} (\text{lazy} \text{ cbody::Expr}) (\text{lazy} \text{ Chandler::Expr})

fixrec \text{eval} :: \text{Expr} \rightarrow \text{Nat Maybe}
where
\text{eval} (\text{Val} n) = \text{Just} \cdot n
| \text{eval} (\text{Add} x y) = \text{mliftM2} (\Lambda a b. a + b) \cdot (\text{eval} x) \cdot (\text{eval} y)
| \text{eval} \cdot \text{Throw} = \text{mfail}
| \text{eval} (\text{Catch} x y) = \text{mcatch} \cdot (\text{eval} x) \cdot (\text{eval} y)
\textbf{fixrec} eval-body :: (Expr \to Nat Maybe) \to Expr \to Nat Maybe
\textbf{where}
\begin{itemize}
\item eval-body \cdot r \cdot (Val \cdot n) = \text{Just} \cdot n
\item eval-body \cdot r \cdot (Add \cdot x \cdot y) = \text{mliftM2} (\Lambda a b. a + b) \cdot (\text{r} \cdot x) \cdot (\text{r} \cdot y)
\item eval-body \cdot r \cdot \text{Throw} = \text{mfail}
\item eval-body \cdot r \cdot (\text{Catch} \cdot x \cdot y) = \text{mcatch} \cdot (\text{r} \cdot x) \cdot (\text{r} \cdot y)
\end{itemize}

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

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

\section{Worker/wrapper}

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

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

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

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

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

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

\textbf{fixrec} eval-body' :: (Expr \to (Nat \to Nat Maybe) \to Nat Maybe \to Nat Maybe)
\to Expr \to (Nat \to Nat Maybe) \to Nat Maybe \to Nat Maybe
\textbf{where}
\begin{itemize}
\item eval-body' \cdot r \cdot (Val \cdot n) \cdot s \cdot f = s \cdot n
\item eval-body' \cdot r \cdot (Add \cdot x \cdot y) \cdot s \cdot f = (\text{case} \ \text{wrapC} \cdot r \cdot x \ \text{of}
\end{itemize}
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 = ⊥
by (subst eval-body ′, unfold, simp)

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

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

lemma eval-body ′ · eval-body-eq: eval-body ′ = unwrapC oo eval-body oo wrapC
apply (intro cfun-eqI)
apply (unfold unwrapC-def wrapC-def)
apply (case-tac xa)
apply simp-all
apply (simp add: wrapC-def)
apply (case-tac x · Expr1 · Just · Nothing)
apply simp-all
apply (case-tac x · Expr2 · Just · Nothing)
apply simp-all
apply (simp add: mfail-def)
apply (simp add: mcatch-def wrapC-def)
apply (case-tac x · Expr1 · Just · Nothing)
apply simp-all
done

fixrec eval-body-final :: (Expr → (Nat → Nat Maybe) → Nat Maybe → Nat Maybe)
→ Expr → (Nat → Nat Maybe) → Nat Maybe → Nat Maybe
where
eval-body-final · r · (Val · n) · s · f = s · n
| eval-body-final · r · (Add · x · y) · s · f = r · x · (Λ n. r · y · (Λ m. s · (n + m)) · f) · f
| eval-body-final · r · Throw · s · f = f
| eval-body-final · r · (Catch · x · y) · s · f = r · x · s · (r · y · s · f)

lemma eval-body-final · strictExpr [simp]: eval-body-final · r · ⊥ · s · f = ⊥
by (subst eval-body-final, unfold, simp)

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

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

definition
  \text{eval-final :: Expr \rightarrow Nat Maybe where}
  \text{eval-final \equiv (\Lambda e. eval-work-final \cdot e \cdot Just \cdot Nothing)}

lemma \text{eval = eval-final}
proof −
  have \text{eval = fix::eval-body by (rule eval-body-eq)}
  also from \text{wrapC-unwrapC-id unwrapC-strict have \ldots = wrapC\cdot(fix::eval-body-final)}
  apply (rule worker-wrapper-fusion-new)
  using \text{eval-body’-eval-body-final-eq eval-body’-eval-body-eq by simp}
  also have \ldots = eval-final
  unfolding \text{eval-final-def eval-work-final-def wrapC-def}
  by simp
  finally show ?thesis .
qed

9 Backtracking using lazy lists and continuations

To illustrate the utility of workerwrapper 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 \text{expr} = \text{const nat} |
  \text{add expr expr} |
  \text{disj expr expr} |
  \text{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 \text{‘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:

\[ \text{fixrec } lappend :: 'a llist \to 'a llist \to 'a llist \text{ where} \]
\[ lappend \cdot lnil \cdot ys = ys \]
\[ lappend \cdot (lcons \cdot x \cdot xs) \cdot ys = lcons \cdot x \cdot (lappend \cdot xs \cdot ys) \]

\[ \text{fixrec } lconcat :: 'a llist llist \to 'a llist \text{ where} \]
\[ lconcat \cdot lnil = lnil \]
\[ lconcat \cdot (lcons \cdot x \cdot xs) = lappend \cdot x \cdot (lconcat \cdot xs) \]

\[ \text{fixrec } lmap :: ('a \to 'b) \to 'a llist \to 'b llist \text{ where} \]
\[ lmap \cdot f \cdot lnil = lnil \]
\[ lmap \cdot f \cdot (lcons \cdot x \cdot xs) = lcons \cdot (f \cdot x) \cdot (lmap \cdot f \cdot xs) \]

We define the lazy list monad \( S \) in the traditional fashion:

\[ \text{type-synonym } S = \text{nat discr llist} \]

\[ \text{definition } returnS :: \text{nat discr} \to S \text{ where} \]
\[ returnS = (\Lambda \cdot x. lcons \cdot x \cdot lnil) \]

\[ \text{definition } bindS :: S \to (\text{nat discr} \to S) \to S \text{ where} \]
\[ bindS = (\Lambda \cdot x g. lconcat \cdot (lmap \cdot g \cdot x)) \]

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:

\[ \text{definition } addS :: S \to S \to S \text{ where} \]
\[ addS \equiv (\Lambda \cdot x y. bindS \cdot x \cdot (\Lambda \cdot x v. bindS \cdot y v \cdot (\Lambda \cdot y v. returnS \cdot x v + y v))) \]

\[ \text{definition } disjS :: S \to S \to S \text{ where} \]
\[ disjS \equiv lappend \]

\[ \text{definition } failS :: S \text{ where} \]
\[ failS \equiv lnil \]

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

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

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

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

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

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

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

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

Our extra constants are defined as follows:

definition \( \text{addK} :: K \rightarrow K \rightarrow K \) 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 \( \text{disjK} :: K \rightarrow K \rightarrow K \) where
\( \text{disjK} \equiv (\Lambda g h. \Lambda s f. g \cdot s \cdot (h \cdot s \cdot f)) \)
definition \( \text{failK} :: K \) where
\( \text{failK} \equiv \Lambda s f. f \)

The continuation semantics is again straightforward:

definition \( \text{evalK-body} :: (\text{expr discr} \rightarrow K) \rightarrow (\text{expr discr} \rightarrow K) \)
where

\[
\text{evalK-body} \equiv \Lambda \, r \, e . \ \text{case undiscr} \ e \ \text{of} \\
\text{const} \ n \Rightarrow \text{returnK} \ (\text{Discr} \ n) \\
| \text{add} \ e1 \ e2 \Rightarrow \text{addK} \ (r \cdot (\text{Discr} \ e1)) \cdot (r \cdot (\text{Discr} \ e2)) \\
| \text{disj} \ e1 \ e2 \Rightarrow \text{disjK} \ (r \cdot (\text{Discr} \ e1)) \cdot (r \cdot (\text{Discr} \ e2)) \\
| \text{fail} \Rightarrow \text{failK}
\]

abbreviation \text{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 \text{unwrap} is the following function that converts a lazy list into an equivalent continuation representation.

\text{fixrec} \ S K :: S \rightarrow K where

\( S K \cdot \text{lnil} = \text{failK} \)
| \( S K \cdot (\text{lcons} \cdot x \cdot xs) = (\Lambda \ s \ f . \ s \cdot x \cdot (S K \cdot xs \cdot s \cdot f)) \)

definition \text{unwrap} :: (\text{expr discr} \rightarrow \text{nat discr list}) \rightarrow (\text{expr discr} \rightarrow K) where

\text{unwrap} \equiv \Lambda \ r \ e . \ S K \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 \rightarrow S where

\( \text{KS} \equiv (\Lambda \ k . \ k \cdot \text{lecons} \cdot \text{lnil}) \)

definition \text{wrap} :: (\text{expr discr} \rightarrow K) \rightarrow (\text{expr discr} \rightarrow \text{nat discr list}) where

\text{wrap} \equiv \Lambda \ r \ e . \ S K \cdot (r \cdot e)

The worker/wrapper condition follows directly from these definitions.

lemma \text{KS-SK-id}:

\( KS \cdot (SK \cdot xs) = xs \)
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 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}:

\( f \cdot x \nsubseteq g \cdot x \Rightarrow f \nsubseteq g \)
by (auto simp: \text{cfun-below-iff})

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

proof –
let \(?witness = \Lambda e. (\Lambda s f. \text{nil} :: K)
have (unwrap oo wrap) \(|witness\) (\text{Discr fail}) \(|\bot\cdot (\text{cons} \cdot \text{nil})
  \| \| ?witness\) (\text{Discr fail}) \(|\bot\cdot (\text{cons} \cdot \text{nil})
by (simp add: failK-def wrap-def unwrap-def KS-def)
hence (unwrap oo wrap) \(!witness \| \| \| ?witness
by (fastforce intro!: cfun-not-below)
thus \(?thesis \| \| (simp add: cfun-not-below)
qed

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

\text{definition eval-work :: expr discr \rightarrow K where}
eval-work \equiv \text{fix}(\text{unwrap oo evalS\_body oo wrap})

\text{definition eval-ww :: expr discr \rightarrow nat discr list where}
eval-ww \equiv \text{wrap\_eval\_work}

\text{lemma evalS = eval-ww}
\text{unfolding eval-ww-def eval-work-def}
\text{using worker\_wrapper\_id[OF wrap\_unwrap\_id]}
\text{by simp}

We now show how the monadic operations correspond by showing that \(SK\) 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.

\text{lemma SK\_returnS\_returnK:}
SK\cdot(\text{returnS}\cdot x) = \text{returnK}\cdot x
by (simp add: returnS-def returnK-def failK-def)

\text{lemma SK\_lappend\_distrib:}
SK\cdot(\text{lappend}\cdot xs\cdot ys)\cdot s\cdot f = SK\cdot xs\cdot s\cdot (SK\cdot ys\cdot s\cdot f)
by (induct xs) (simp-all add: failK-def)

\text{lemma SK\_bindS\_bindK:}
SK\cdot(\text{bindS}\cdot x\cdot g) = \text{bindK}\cdot(SK\cdot x)\cdot (SK\cdot oo g)
by (induct x)
(simp-all add: cfun-eq-iff
  bindS-def bindK-def failK-def
  SK\_lappend\_distrib)

\text{lemma SK\_addS\_distrib:}
SK\cdot(\text{addS}\cdot x\cdot y) = \text{addK}\cdot(SK\cdot x)\cdot (SK\cdot y)
by (clarsimp simp: cfcomp1
  addS-def addK-def failK-def
  SK\_bindS\_bindK SK\_returnS\_returnK)

\text{lemma SK\_disjS\_disjK:}
\[ SK \cdot (\text{disj} \cdot x \cdot y) = \text{disj}K \cdot (SK \cdot x) \cdot (SK \cdot y) \]

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

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

proof (intro cfun-eqI)

fix \( r \) \( e' \) \( s \) \( f \)

obtain \( e \) :: expr where \( ee' = \text{Discr e} \) by (cases e')

have \( \text{(unwrap oo evalS-body oo wrap)} \cdot r \cdot (\text{Discr e} \cdot s \cdot f) \)

by (cases e)

(simp-all add: evalS-body-def evalK-body-def unwrap-def
  SK-returnS-returnK SK-adds-distrib
  SK-disjS-disjK SK-failS-failK)

with \( ee' \) show \( \text{(unwrap oo evalS-body oo wrap)} \cdot r \cdot e' \cdot s \cdot f \)

by simp

qed

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

using worker-wrapper-fusion-new[OF wrap-unwrap-id unwrap-strict]

  evalS-body-evalK-body

by 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 \texttt{nub} function.

\texttt{fixrec nub :: Nat list \rightarrow Nat list}
where
\begin{verbatim}
nub-nil = lnil
\mid nub-(x :@ xs) = x :@ nub-(lfilter-(neg oo (\Lambda y. x =_B y)) \cdot xs)
\end{verbatim}

\texttt{lemma nub-strict[simp]}: \texttt{nub-\bot = \bot}
by \texttt{fixrec-simp}

\texttt{fixrec nub-body :: (Nat list \rightarrow Nat list) \rightarrow Nat list \rightarrow Nat list}
where
\begin{verbatim}
nub-body-f-nil = lnil
\mid nub-body-f-(x :@ xs) = x :@ f-(lfilter-(neg oo (\Lambda y. x =_B y)) \cdot xs)
\end{verbatim}

\texttt{lemma nub-nub-body-eq}: \texttt{nub = fix \cdot nub-body}
by \texttt{(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}
\begin{verbatim}
SetEmpty :: NatSet where
SetEmpty \equiv lnil
\end{verbatim}

\texttt{definition}
\begin{verbatim}
SetInsert :: Nat \rightarrow NatSet \rightarrow NatSet where
SetInsert \equiv lcons
\end{verbatim}

\texttt{definition}
\begin{verbatim}
SetMem :: Nat \rightarrow NatSet \rightarrow tr where
SetMem \equiv lmember-(bpred (=))
\end{verbatim}

\texttt{lemma SetMem-strict[simp]}: \texttt{SetMem-x-\bot = \bot}
by \texttt{(simp add: SetMem-def)}

\texttt{lemma SetMem-SetEmpty[simp]}: \texttt{SetMem-x-SetEmpty = FF}
by \texttt{(simp add: SetMem-def SetEmpty-def)}

\texttt{lemma SetMem-SetInsert}: \texttt{SetMem-v-(SetInsert-x-s) = (SetMem-v-s orelse x =_B v)}
by \texttt{(simp add: SetMem-def SetInsert-def)}

AndyG’s new type.

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

\texttt{definition}
\begin{verbatim}
nextR :: R \rightarrow (Nat * R) Maybe where
nextR = (\Lambda r. case ldropWhile-(\Lambda x. SetMem-x-(exceptR-r))-(resultR-r) of
\end{verbatim}
\[\text{lnil} \Rightarrow \text{Nothing} \quad | \quad x : @ \text{xs} \Rightarrow \text{Just}(x, R\cdot\text{xs} \cdot (\text{exceptR} \cdot r))\]

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

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

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

**Definition**

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

**Definition**

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

**Definition**

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

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

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

by \((\text{rule cfun-eqI}, \text{simp add: a2c-def c2a-def lfilter-const-true})\)

**Definition**

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

**Definition**

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

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

unfolding \(\text{unwrap-def}\) by \((\text{rule cfun-eqI}, \text{simp})\)

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

using \(\text{cfun-fun-cong[OF a2c-c2a-id]}\)

by - \((\text{rule cfun-eqI} +, \text{simp add: wrap-def unwrap-def})\)

Equivalences needed for later.

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

by \((\text{rule trE[where p=x], simp-all})\)

**Lemma** case-maybe-case:

\[
\begin{align*}
\text{case} \quad \text{case L of lnil} \Rightarrow \text{Nothing} \quad | \quad x : @ \text{xs} \Rightarrow \text{Just}(h \cdot x \cdot xs) \quad \text{of} \\
\text{Nothing} \Rightarrow f \\
\text{Just}(a, b) \Rightarrow g \cdot a \cdot b
\end{align*}
\]

\[
\begin{align*}
\text{case} \quad \text{case L of lnil} \Rightarrow f \\
\text{case} \quad x : @ \text{xs} \Rightarrow g \cdot (\text{fst} \cdot (h \cdot x \cdot xs)) \cdot (\text{snd} \cdot (h \cdot x \cdot xs))
\end{align*}
\]

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

lemma case-a2c-case-caseR:
  (case a2c-w of lnil ⇒ f | x :@ xs ⇒ g·x·xs)
  = (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·xs·(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·xs·(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·xs·(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 =\_ y))·(a2c·r) = a2c·(filterR·x·r)
using filter-filter [where p=Tr·neg oo (Λ y. x =\_ 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 =\_ 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{(SetMem v{(exceptR xa)})})(resultR xa)
    , 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·(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=lnil 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 llast-strict[simp]: llast·⊥ = ⊥
by fixrec-simp

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

lemma llast-llast-body: llast = fix·llast-body
by (rule cfun-eqI, subst llast-def, subst llast-body, unfold, simp)

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

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

lemma unwrap-strict[simp]: unwrap·⊥ = ⊥
unfolding unwrap-def by ((rule cfun-eqI)+, simp)

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

definition llast-worker :: ('a → 'a llist → 'a) → 'a → 'a llist → 'a where
llast-worker ≡ Λ r·x@yys. case yys of lnil ⇒ x | y@ys ⇒ r·y·ys

definition llast′ :: 'a list → 'a llist → 'a where
llast′ ≡ wrap·(fix·llast-worker)

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

lemma llast′-llast: llast′ = llast (is ?lhs = ?rhs)
proof
have ?rhs = fix·llast-body by (simp only: llast-llast-body)
also have ... = wrap·(fix·(unwrap oo llast-body oo wrap))
  by (simp only: worker-wrapper-body[OF wrap-unwrap-ID])
also have ... = wrap·(fix·(llast-worker))
  by (simp only: llast-worker-llast-body)
also have ... = ?lhs unfolding llast′-def by simp
finally show ?thesis by 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 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


