Mechanising the worker/wrapper transformation

Peter Gammie

April 20, 2020

Contents

1 Introduction

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

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

4 A totally-correct fusion rule

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

6 Unboxing types.
   6.1 Factorial example
   6.2 Introducing an accumulator

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

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

9 Backtracking using lazy lists and continuations
10 Transforming $O(n^2)$ nub into an $O(n \lg n)$ one
  10.1 The nub function. .................................................. 34
  10.2 Optimised data type. ................................................. 35

11 Optimise “last”.
  11.1 The 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 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 recrusive
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).

\begin{spverbatim}
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))
\end{spverbatim}
using \texttt{fix-least-below} by \texttt{blast} — induction

\textbf{qed}

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

\textbf{proof} —
\hspace{0.5em} have \( \text{fix} \cdot (f \circ g) \sqsubseteq f \cdot (\text{fix} \cdot (g \circ f)) \) by (rule rolling-rule-ltr)
\hspace{0.5em} hence \( g \cdot (\text{fix} \cdot (f \circ g)) \sqsubseteq g \cdot (f \cdot (\text{fix} \cdot (g \circ f))) \)
\hspace{0.5em} by (rule monofun-cfun-arg) — \( g \) is monotonic
\hspace{0.5em} thus \( g \cdot (\text{fix} \cdot (f \circ g)) \sqsubseteq \text{fix} \cdot (g \circ f) \)
\hspace{0.5em} using \texttt{fix-eq}[\text{where } F=g \circ f] by \texttt{simp} — computation

\textbf{qed}

\textbf{lemma} rolling-rule: \( \text{fix} \cdot (g \circ f) = g \cdot (\text{fix} \cdot (f \circ g)) \)
\hspace{0.5em} by (rule below-antisym[OF rolling-rule-ltr rolling-rule-rtl])

\section*{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 (f) at (0,0) [circle, draw] {$f$};
  \node (g) at (1,1) [circle, draw] {$g$};
  \node (h) at (2,2) [circle, draw] {\text{fix } h};
  \node (g') at (1,0) [circle, draw] {$g$};
  \node (h') at (2,1) [circle, draw] {\text{fix } f};
  \path (f) edge (g); % f \rightarrow g
  \path (g) edge (h); % g \rightarrow \text{fix } h
  \path (g') edge (g); % g \rightarrow g
  \path (g') edge (h'); % g \rightarrow \text{fix } f
  \path (h) edge (h'); % \text{fix } h \rightarrow \text{fix } f
\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 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}:

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

in a very straightforward way:

\textbf{lemma} lfp-fusion:
\hspace{0.5em} \text{assumes } g \perp = \perp
\hspace{0.5em} \text{assumes } g \circ f = h \circ g
\hspace{0.5em} \text{shows } g \cdot (\text{fix } f) = \text{fix } h

\textbf{proof}(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:

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

(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).

```
case 2 show \( g \cdot \perp = \perp \) by fact
case (3 x y)
from \( g \cdot x = y \) \( \cdot g \circ f = h \circ g \) show \( g \cdot (f \cdot x) = h \cdot y \)
by (simp add: cfun-eq-iff)
qed simp
```

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.

```
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 \( \text{computation} = \text{wrap} \cdot (\text{fix} \cdot (\text{unwrap} \circ \text{body} \circ oo \text{wrap})) \)
proof –
  from \( \text{comp-body} \) have \( \text{computation} = \text{fix} \cdot (\text{id} \circ \text{body}) \)
```

4
by simp
also from wrap-unwrap have \ldots\ = \texttt{fix}(\texttt{wrap oo unwrap oo body})
by (simp add: assoc-oo)
also have \ldots\ = \texttt{wrap}.(\texttt{fix}(\texttt{unwrap oo body oo wrap}))
  using rolling-rule[where f=unwrap oo body and g=wrap]
  by (simp add: assoc-oo)
finally show \texttt{thesis}.
qed

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

\textbf{lemma} \texttt{worker-wrapper-body}:
\begin{itemize}
\item fixes \texttt{wrap} :: 'b::pcpo \to 'a::pcpo
\item fixes \texttt{unwrap} :: 'a \to 'b
\item assumes \texttt{wrap-unwrap}: \texttt{wrap oo unwrap oo body} = \texttt{body}
\item assumes \texttt{comp-body}: \texttt{computation} = \texttt{fix-body}
\end{itemize}
shows \texttt{computation} = \texttt{wrap}.(\texttt{fix}(\texttt{unwrap oo body oo wrap}))
\texttt{proof}
\begin{itemize}
\item from \texttt{comp-body} have \texttt{computation} = \texttt{fix}(\texttt{wrap oo unwrap oo body})
  using \texttt{wrap-unwrap} by (simp add: assoc-oo \texttt{wrap-unwrap})
\item also have \ldots\ = \texttt{wrap}.(\texttt{fix}(\texttt{unwrap oo body oo wrap}))
  using rolling-rule[where f=unwrap oo body and g=wrap]
  by (simp add: assoc-oo)
\end{itemize}
finally show \texttt{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.

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

Gill and Hutton's \texttt{worker-wrapper-fusion} rule is intended to allow the transformation of \texttt{(unwrap oo wrap)}-\texttt{R} to \texttt{R} in recursive contexts, where \texttt{R} is meant to be a self-call. Note that it assumes that the first worker/wrapper
hypothesis can be established.

**Lemma worker-wrapper-fusion:**

- **fixes** \( \text{wrap} : 'b::\text{pcpo} \to 'a::\text{pcpo} \)**
- **fixes** \( \text{unwrap} : 'a \to 'b \)**
- **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** \( (\text{unwrap} \circ \text{wrap}) \circ \text{work} = \text{work} \)

**proof**

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

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

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

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

- **finally show** \( ?\text{thesis using} \) \( \text{work} \) **by** \( \text{simp} \)

**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 \( \text{body} \implies f \), one can always derive \( f = f \). Clearly this has dire implications for the preservation of termination behaviour.

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

\[
\frac{f \cdot ?x = ?x}{\text{fix}_x f \subseteq ?x} \quad \text{fix\_least}
\]
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. $\text{unwrap} \not\equiv \bot$) 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 $\text{unwrap oo body oo wrap}$, can be rewritten as $\text{body}' \cdot \text{unwrap oo body oo wrap}$. 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:

$$\text{fix} \cdot (\text{unwrap oo body oo wrap}) = \text{fix} \cdot \text{body}'$$

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

**proof** (rule fix-least)
- have $\text{work} = (\text{unwrap oo body oo wrap}) \cdot \text{work}$
  - using $\text{work}$ by (simp add: fix-eq[symmetric])
- also have ... $= (\text{body}' \cdot \text{unwrap oo wrap}) \cdot \text{work}$
  - using $\text{body}'$ by simp
- also have ... $= (\text{body}' \cdot \text{unwrap oo wrap}) \cdot (\text{unwrap oo body oo wrap}) \cdot \text{work}$
  - using $\text{work}$ by (simp add: fix-eq[symmetric])
- also have ... $= (\text{body}' \cdot \text{unwrap oo wrap} \cdot \text{unwrap oo body oo wrap}) \cdot \text{work}$
  - by simp
- also have ... $= (\text{body}' \cdot \text{unwrap oo body oo wrap}) \cdot \text{work}$
  - using 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** $A = A$
domain $B = B$ (lazy $A$)

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

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

fixrec unwrap :: $A \rightarrow B$
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 :: $A \rightarrow A$
where body$\cdot r = A$
\end{verbatim}

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

\begin{verbatim}
fixrec body' :: $B \rightarrow B$
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 \texttt{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 \texttt{unwrap} is not strict. In the following section we show that requiring \texttt{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 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 `worker-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 `worker-wrapper-id` and the ubiquitous `lfp-fusion` rule.

**lemma** `worker-wrapper-fusion-new`:
- fixes \( \text{wrap} :: 'b::pcpo \rightarrow 'a::pcpo \)
- fixes \( \text{unwrap} :: 'a \rightarrow 'b \)
- fixes `body' :: 'b \rightarrow 'b`
- assumes `wrap-unwrap`: \( \text{wrap} \circ \text{unwrap} = (\text{ID} :: 'a \rightarrow 'a) \)
- assumes `unwrap-strict`: \( \text{unwrap} \cdot \bot = \bot \)
- assumes `body-body'`: \( \text{unwrap} \circ \text{body} \circ \text{unwrap} = \text{body}' \circ \text{unwrap} \circ \text{unwrap} \)
- shows `fix-body = \text{wrap} \cdot (\text{fix-body}')`

**proof**
- from `body-body'`
  - have \( \text{unwrap} \circ \text{body} \circ (\text{wrap} \circ \text{unwrap}) = (\text{body}' \circ \text{unwrap} \circ \text{unwrap} \circ (\text{wrap} \circ \text{unwrap})) \)
    - by (simp add: `assoc-oo`)
  - with `wrap-unwrap` have \( \text{unwrap} \circ \text{body} \circ \text{unwrap} = \text{body}' \circ \text{unwrap} \circ \text{unwrap} \)
    - by `simp`
  - with `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 `wrap-unwrap` show `?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 \( comp = body \) of type \( A \) and a pair of functions \( wrap : B \to A \) and \( unwrap : A \to B \) where \( wrap \circ unwrap = id_A \) and \( unwrap \bot = \bot \), define:

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

(the worker/wrapper transformation)

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

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

(worker/wrapper fusion)

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

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

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

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

**definition**

\[
\begin{align*}
\text{H2list} &:: a H \to a \text{llist} \\
\text{H2list} &\equiv \Lambda f. f \cdot \text{lnil}
\end{align*}
\]

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

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

**proof** (rule cfun-eqI)

fix \( zs \)

have \( ?lhs \cdot zs = (xs :++ ys) :++ zs \) 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 \( \ldots = (\text{list2H} \cdot xs \circ \text{list2H} \cdot ys) \cdot zs \) by simp

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

qed

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

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

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

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

**fixrec** \( \text{lrev} :: a \text{llist} \to a \text{llist} \)
where
\[
\text{rev-nil} = \text{nil}
\]
| \text{rev}(x :@ xs) = rev \cdot xs :++ (x :@ \text{nil})

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

**Lemma** \text{rev-strict(simp)}: \text{rev} \cdot \bot = \bot

by \text{fixrec-simp}

**Fixrec** \text{rev-body} :: (\text{’a list} \to \text{’a list}) \to \text{’a list}

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

**Lemma** \text{rev-body-strict(simp)}: \text{rev-body} \cdot r \cdot \bot = \bot

by \text{fixrec-simp}

This is trivial but syntactically a bit touchy. Would be nicer to define \text{rev-body} as the generator of the fixpoint definition of \text{rev} directly.

**Lemma** \text{rev-rev-body-eq}: \text{rev} = \text{fix} \cdot \text{rev-body}

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

Wrap / unwrap functions.

**Definition**
\text{unwrapH} :: (\text{’a list} \to \text{’a list}) \to \text{’a list} \to \text{’a H}

where
\[
\text{unwrapH} \equiv \Lambda f \cdot x . \text{list2H} \cdot (f \cdot x)
\]

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

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

**Definition**
\text{wrapH} :: (\text{’a list} \to \text{’a H}) \to \text{’a list} \to \text{’a list}

where
\[
\text{wrapH} \equiv \Lambda f \cdot x . \text{H2list} \cdot (f \cdot x)
\]

**Lemma** \text{wrapH-unwrapH-id}: \text{wrapH} oo \text{unwrapH} = \text{ID} (is \ ?lhs = \ ?rhs)

**Proof** (rule \text{cfun-eqI}) +

fix \ f \ x

have \ ?lhs \cdot f \cdot x = \text{H2list} \cdot \text{list2H} \cdot (f \cdot x) by (simp add: wrapH-def unwrapH-def)

also have \ \ldots = (\text{H2list clean list2H}) \cdot (f \cdot x) by simp

also have \ \ldots = \text{ID} \cdot (f \cdot x) by (simp only: H2list-list2H-inv)

also have \ \ldots = \ ?rhs \cdot f \cdot x by simp

finally show \ ?lhs \cdot f \cdot x = \ ?rhs \cdot f \cdot x .

qed

5.2 Gill/Hutton-style worker/wrapper.

**Definition**
\text{rev-work} :: \text{’a list} \to \text{’a H}

\text{rev-work} \equiv \text{fix} \cdot (\text{unwrapH} oo \text{rev-body} oo \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).
unfold
apply (subst lrev-body1.
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
lemma lrev-body2-lrev-body1-eq: lrev-body2 = lrev-body1
  by ((rule cfun-eqI)+
    , (subst lrev-body1.unfold, subst lrev-body2.unfold)
    , (simp add: H-list-hom-append[symmetric] H-list-hom-id))

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)
    done
  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
  }
next
case \texttt{\texttt{lnil}} thus \?thesis
  unfolding \texttt{lrev-work2-def}
  by (subst \texttt{fix-eq[where } \texttt{F=lrev-body2], simp})
next
case (\texttt{lcons \texttt{y} \texttt{ys})}
  hence \texttt{lrev-body3·lrev-work2·xs = lrev-work2·ys oo list2H·(y :@ lnil)} by simp
  also have \ldots = \texttt{list2H·((wrapH·lrev-work2)·ys) oo list2H·(y :@ lnil)}
  using \texttt{lrev-wafusion[where } \texttt{xs=ys]} by simp
  also from \texttt{lcons} have \ldots = \texttt{lrev-body2·lrev-work2·xs} by simp
  also have \ldots = \texttt{lrev-work2·xs}
  unfolding \texttt{lrev-work2-def} by (simp only: \texttt{fix-eq[symmetric]})
  finally show \?thesis by simp
qed

thus \texttt{lrev-body3·lrev-work2 = lrev-work2} by (rule \texttt{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 \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 com-
pletely sound in this context, by appealing to the lazy list induction princi-
ple.

\textbf{lemma} \texttt{lrev-work3-lrev-work2-eq: lrev-work3 = lrev-work2} (is \texttt{?lhs = ?rhs})
\textbf{proof}(rule \texttt{cfun-eqI})
  fix \texttt{x}
  show \texttt{?lhs·x = ?rhs·x}
  proof (induct \texttt{x})
  show \texttt{lrev-work3·\bot = lrev-work2·\bot}
    apply (unfold \texttt{lrev-work3-def} \texttt{lrev-work2-def})
    apply (subst \texttt{fix-eq[where } \texttt{F=lrev-body2]})
    apply (subst \texttt{fix-eq[where } \texttt{F=lrev-body3]})
    by (simp add: \texttt{lrev-body3.unfold lrev-body2.unfold})
  next
  show \texttt{lrev-work3·\texttt{lnil} = lrev-work2·\texttt{lnil}}
    apply (unfold \texttt{lrev-work3-def} \texttt{lrev-work2-def})
    apply (subst \texttt{fix-eq[where } \texttt{F=lrev-body2]})
    apply (subst \texttt{fix-eq[where } \texttt{F=lrev-body3]})
    by (simp add: \texttt{lrev-body3.unfold lrev-body2.unfold})
  next
  fix \texttt{a l} assume \texttt{lrev-work3·a = lrev-work2·a}
  thus \texttt{lrev-work3·(a :@ l) = lrev-work2·(a :@ l)}
    apply (unfold \texttt{lrev-work3-def} \texttt{lrev-work2-def})
    apply (subst \texttt{fix-eq[where } \texttt{F=lrev-body2]})
    apply (subst \texttt{fix-eq[where } \texttt{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 (subl 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 ... = 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 list → 'a H) → 'a list → 'a H
where
  lrev-body-final·r·nil·ys = ys
| lrev-body-final·r·(x :@ xs)·ys = r·xs·(x :@ ys)

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

definition
  lrev-final :: 'a list → 'a list where
  lrev-final ≡ Λ xs. lrev-work-final·xs·nil

lemma lrev-body-final-lrev-body3-eq': lrev-body-final·r·xs = lrev-body3·r·xs
apply (subl lrev-body-final.unfold)
apply (subl 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 (auto)
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 ... = 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 (>>=) 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'-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) >>= r·(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

19
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-work-def)
  also have ... = fac-final by (simp add: fac-final-def wrapB-def)
final 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
\( \text{wrapA} :: (\text{UNat} \rightarrow \text{UNatAcc}) \rightarrow \text{UNat} \rightarrow \text{UNat} \rightarrow \text{UNat} \) where
\( \text{wrapA} \equiv \Lambda f.~a2n~oo~f \)

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
\( \text{fac-acc-body1} :: (\text{UNat} \rightarrow \text{UNatAcc}) \rightarrow \text{UNat} \rightarrow \text{UNatAcc} \) where
\( \text{fac-acc-body1} \equiv \Lambda r~n.\quad \text{if } n = 0 \text{ then } n2a \cdot 1 \text{ else } \text{wrapA} \cdot r \cdot (n - \# 1) >\Rightarrow (\Lambda \text{res}.~n2a \cdot (n *\# \text{res})) \)

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
\( \text{fac-acc-body2} :: (\text{UNat} \rightarrow \text{UNatAcc}) \rightarrow \text{UNat} \rightarrow \text{UNatAcc} \) where
\( \text{fac-acc-body2} \equiv \Lambda r~n.\quad \text{if } n = 0 \text{ then } n2a \cdot 1 \text{ else } \text{wrapA} \cdot r \cdot (n - \# 1) >\Rightarrow (\Lambda \text{res}.~n2a \cdot n >\Rightarrow n2a \cdot \text{res}) \)

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
\( \text{fac-acc-body3} :: (\text{UNat} \rightarrow \text{UNatAcc}) \rightarrow \text{UNat} \rightarrow \text{UNatAcc} \) where
\( \text{fac-acc-body3} \equiv \Lambda r~n.\quad \text{if } n = 0 \text{ then } n2a \cdot 1 \text{ else } n2a \cdot n >\Rightarrow r \cdot (n - \# 1) \)

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=\Lambda y.~n2a·n >\Rightarrow 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-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·(smap·f·xs) = smap·(f oo g)·xs
fixrec i-th :: 'a Stream → Nat → 'a
where

22
\[ i-th(x \&\& xs) = Nat-case_\times (i-th\times s) \]

**abbreviation**

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

The infinite stream of natural numbers.

**fixrec** \( nats :: Nat \text{ Stream} \)
**where**
\[ nats = 0 \&\& \text{smap}(\Lambda x. 1 + x) \cdot nats \]

### 7.2 The wrapper/unwrapper functions.

**definition**

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

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

**fixrec** \( \text{unwrapS} :: (Nat \rightarrow 'a) \rightarrow 'a \text{ Stream} \)
**where**
\[ \text{unwrapS} \cdot f = f\cdot 0 \&\& \text{unwrapS}.(f \circ (\Lambda x. 1 + x)) \]

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** \( \text{unwrapS-unwrapS}'\text{-eq} : \text{unwrapS} = \text{unwrapS}' \text{ (is } ?lhs = ?rhs) \)

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

fix \( f \) show \( ?lhs\cdot f = ?rhs\cdot f \)

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

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

show \( \text{Stream-bisim} \ ?\text{R} \)

**proof**

fix \( s s' \) assume \( ?\text{R} \cdot s \cdot s' \)
then obtain \( f \) where \( fs: \ s = f\cdot 0 \&\& \text{unwrapS}.(f \circ (\Lambda x. 1 + x)) \)
and \( fs': s' = f\cdot 0 \&\& \text{smap}(f \circ (\Lambda x. 1 + x))\cdot nats \)

by blast

have \( ?\text{R} \cdot (\text{unwrapS}.(f \circ (\Lambda x. 1 + x))) \cdot (\text{smap}(f \circ (\Lambda x. 1 + x))\cdot nats) \)
by ( rule extI [where \( x = f \circ (\Lambda x. 1 + x) \])
, subst \text{unwrapS}.unfold, subst nats.unfold, simp add: \text{smap-smap} )
with \( fs \cdot fs' \)

show \( (s = \bot \&\& s' = \bot) \)
\[ \lor (\exists h t \ t'. \ (\exists f. t = f\cdot 0 \&\& \text{unwrapS}.(f \circ (\Lambda x. 1 + x)) \&\& t' = f\cdot 0 \&\& \text{smap}(f \circ (\Lambda x. 1 + x))\cdot nats) \&\& s = h \&\& t \&\& s' = h \&\& t' ) \text{ by best} \]

qed
show ?R (?lhs·f) (?rhs·f)
proof –

have lhs: ?lhs·f = f·0 &\& unwrapS·(f oo (Λ x. 1 + x)) by (subst unwrapS·unfold, simp)

have rhs: ?rhs·f = f·0 &\& smap·(f oo (Λ x. 1 + x))·nats by (rule unwrapS·'·unfold)

from lhs rhs show ?thesis by best
qed

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::Nat → 'a)·⊥ = ⊥

shows unwrapS·f !! n = f·n

using strictF

proof(induct n arbitrary: f rule: Nat-induct)

next

case bottom with strictF show ?case by simp

next

case zero thus ?case by (subst unwrapS·unfold, simp)

next

case (Suc i f)

have unwrapS·f !! (i + 1) = (f·0 &\& unwrapS·(f oo (Λ x. 1 + x))) !! (i + 1)

by (subst unwrapS·unfold, simp)

also from Suc have \( \ldots = unwrapS·(f oo (Λ x. 1 + x)) \) !! i by simp

also from Suc have \( \ldots = (f oo (Λ x. 1 + x))·i \) by simp

also have \( \ldots = f·(i + 1) \) by (simp add: plus-commute)

finally show ?case .

qed

lemma wrapS-unwrapS-id: f·⊥ = ⊥ \( \Rightarrow (wrapS oo unwrapS)·f = f \)

by (rule cfun-eqI, simp add: wrapS-unwrapS-id' wrapS-def)

7.3 Fibonacci example.

definition

\[ \text{fib-body} :: (Nat → Nat) → Nat → Nat \]

where

\[ \text{fib-body} ≡ Λ r. \text{Nat-case}·1·(\text{Nat-case}·1·(Λ n. r·n + r·(n + 1))) \]

definition

\[ \text{fib} :: Nat → Nat \]

where

\[ \text{fib} ≡ \text{fix}·\text{fib-body} \]
Apply worker/wrapper.

**definition**

\[
\text{fib-work} :: \text{Nat Stream where}
\]

\[
\text{fib-work} \equiv \text{fix} \cdot (\text{unwrapS oo fib-body oo wrapS})
\]

**definition**

\[
\text{fib-wrap} :: \text{Nat} \rightarrow \text{Nat where}
\]

\[
\text{fib-wrap} \equiv \text{wrapS} \cdot \text{fib-work}
\]

**lemma** wrapS-unwrapS-fib-body:

\[
\text{wrapS oo unwrapS oo fib-body} = \text{fib-body}
\]

**proof**

\[
\text{rule cfun-eqI}
\]

\[
\text{fix } r \text{ show } (\text{wrapS oo unwrapS oo fib-body}) \cdot r = \text{fib-body} \cdot r
\]

**using** wrapS-unwrapS-id[where \(f=\text{fib-body} \cdot r\)] by simp

**qed**

**lemma** fib-ww-eq:

\[
\text{fib} = \text{fib-wrap}
\]

**using** worker-unwrap-body[\(\text{OF wrapS-unwrapS-fib-body}\)]

**by** (simp add: fib-def fib-wrap-def fib-work-def)

Optimise.

**fixrec**

\[
\text{fib-work-final} :: \text{Nat Stream}
\]

and

\[
\text{fib-f-final} :: \text{Nat} \rightarrow \text{Nat where}
\]

\[
\text{fib-work-final} = \text{smap} \cdot \text{fib-f-final} \cdot \text{nats}
\]

\[
| \text{fib-f-final} = \text{Nat-case} \cdot 1 \cdot (\text{Nat-case} \cdot 1 \cdot (\Lambda n'. \text{fib-work-final} !! n' + \text{fib-work-final} !! (n' + 1)))
\]

**declare** fib-f-final.simps[simp del] fib-work-final.simps[simp del]

**definition**

\[
\text{fib-final} :: \text{Nat} \rightarrow \text{Nat where}
\]

\[
\text{fib-final} \equiv \Lambda n. \text{fib-work-final} !! n
\]

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:

\[
\text{fib-work-final} = \text{fib-work} \text{ (is } ?lhs = ?rhs)
\]

**proof**

\[
\text{let } ?wb = \Lambda r. \text{Nat-case} \cdot 1 \cdot (\text{Nat-case} \cdot 1 \cdot (\Lambda n'. r !! n' + r !! (n' + 1)))
\]

\[
\text{let } ?mr = \Lambda (\text{ff} :: \text{Nat Stream, fff}). (\text{smap} \cdot \text{ff} \cdot \text{nats}, ?wb \cdot \text{ff})
\]

\[
\text{have } ?lhs = \text{fst} (\text{fix} \cdot ?mr)
\]

**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 \( \ldots = (\mu\ f\ f,\ \text{fst}\ (\mu mr\cdot fuf,\ \mu fff.\ \text{snd}\ (\mu mr\cdot fuf,\ fff))) \)
using \( \text{fix-cprod}[\text{where} \: F=\mu mr] \) by simp
also have \( \ldots = (\mu\ fuf,\ \text{smap}(\mu fff.\ wb\cdot fuf)\cdot\text{nats}) \) by simp
also have \( \ldots = (\mu\ fuf.\ \text{smap}(\mu wb\cdot fuf)\cdot\text{nats}) \) by (simp add: \text{fix-const})
also have \( \ldots = \text{?rhs} \)
unfolding \( \text{fib-body-def fib-def unwraps-unwraps'}-\text{eq unwraps'}-\text{def wrapS-def} \)
by (simp add: cfcomp1)
finally show \( \text{?thesis} \).
qed

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

8 Tagless interpreter via double-barreled continuations

type-synonym \( 'a\ Cont = (\: 'a \rightarrow \: 'a) \rightarrow 'a \)
definition \( \text{val2cont} :: \: 'a \rightarrow 'a\ Cont \) where
\( \text{val2cont} \equiv (\Lambda a\ c.\ c\cdot a) \)
definition \( \text{cont2val} :: 'a\ Cont \rightarrow 'a \) where
\( \text{cont2val} \equiv (\Lambda f.\ f\cdot\text{ID}) \)
lemma \( \text{cont2val-val2cont-id} \): \( \text{cont2val oo val2cont} = \text{ID} \)
by (rule \text{cfun-eqI}, simp add: \text{val2cont-def cont2val-def})
domain \( \text{Expr} = \)
\( \text{Val} (\text{lazy} \: \text{val::Nat}) \)
\( \mid \text{Add} (\text{lazy} \: \text{addl::Expr}) (\text{lazy} \: \text{addr::Expr}) \)
\( \mid \text{Throw} \)
\( \mid \text{Catch} (\text{lazy} \: \text{cbody::Expr}) (\text{lazy} \: \text{chandler::Expr}) \)
fixrec \( \text{eval} :: \: \text{Expr} \rightarrow \text{Nat} \) Maybe
where
\( \text{eval} (\text{Val}\: n) = \text{Just}\: n \)
\( \mid \text{eval} (\text{Add}\: x\cdot y) = \text{mliftM2} (\: \Lambda a\: a\cdot b\cdot (\text{eval}\: x)\cdot (\text{eval}\: y)) \)
\( \mid \text{eval} \: \text{Throw} = \text{mfail} \)
\( \mid \text{eval} (\text{Catch}\: x\cdot y) = \text{mcatch} (\text{eval}\: x) (\text{eval}\: y) \)
\textbf{fixrec} eval-body :: (Expr → Nat Maybe) → Expr → Nat Maybe

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*}

\textbf{lemma} eval-body-strictExpr[simp]: eval-body \cdot r \cdot \bot = \bot
\hspace{1cm} \text{by (subst eval-body, unfold, simp)}

\textbf{lemma} eval-eval-body-eq: eval = fix \cdot eval-body
\hspace{1cm} \text{by (rule cfun-eqI, subst eval-def, subst eval-body, unfold, simp)}

\subsection*{8.1 Worker/wrapper}

\textbf{definition} unwrapC :: (Expr → Nat Maybe) → (Expr → (Nat → Nat Maybe) → Nat Maybe) → Nat Maybe
\hspace{1cm} \text{where}
\hspace{1cm} \text{unwrapC} \equiv \Lambda g \ e \ s \ f. \ \text{case} \ g \cdot e \ \text{of} \ \text{Nothing} \Rightarrow f \mid \text{Just} \cdot n \Rightarrow s \cdot n

\textbf{lemma} unwrapC-strict[ simp]: unwrapC \cdot \bot = \bot
\hspace{1cm} \text{unfolding unwrapC-def by (rule cfun-eqI)+ simp}

\textbf{definition} wrapC :: (Expr → (Nat → Nat Maybe) → Nat Maybe → Nat Maybe) → (Expr → Nat Maybe)
\hspace{1cm} \text{where}
\hspace{1cm} \text{wrapC} \equiv \Lambda g \ e. \, g \cdot e \cdot \text{Just} \cdot \text{Nothing}

\textbf{lemma} wrapC-unwrapC-id: wrapC oo unwrapC = ID
\hspace{1cm} \text{proof}\,(\text{intro cfun-eqI})
\hspace{1cm} \text{fix} \ g \ e
\hspace{1cm} \text{show} (\text{wrapC} oo \text{unwrapC}) \cdot g \cdot e = ID \cdot g \cdot e
\hspace{1cm} \hspace{1cm} \text{by (cases g \cdot e, simp-all add: wrapC-def unwrapC-def)}
\hspace{1cm} \text{qed}

\textbf{definition} eval-work :: Expr → (Nat → Nat Maybe) → Nat Maybe → Nat Maybe
\hspace{1cm} \text{where}
\hspace{1cm} \text{eval-work} \equiv \text{fix} \cdot (\text{unwrapC} oo \text{eval-body} oo \text{wrapC})

\textbf{definition} eval-wrap :: Expr → Nat Maybe
\hspace{1cm} \text{where}
\hspace{1cm} \text{eval-wrap} \equiv \text{wrapC} \cdot \text{eval-work}

\textbf{fixrec} eval-body’ :: (Expr → (Nat → Nat Maybe) → Nat Maybe → Nat Maybe) → Expr → (Nat → Nat Maybe) → Nat Maybe → Nat Maybe
\hspace{1cm} \text{where}
\hspace{1cm} \text{eval-body’} \cdot r \cdot (\text{Val} \cdot n) \cdot s \cdot f = s \cdot n
\hspace{1cm} | \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}
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 (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 wrapC-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 =
By relaxing the default sort of type variables to \textit{predomain}, our polymorphic definitions can be used at concrete types that do not contain $\perp$. These include those constructed from HOL types using the discrete ordering type constructor $'a\ discr$, and in particular our interpretation $\text{nat\ discr}$ of the natural numbers.

The following standard list functions underpin the monadic infrastructure:

\begin{verbatim}
fixrec lappend :: 'a llist \to 'a llist \to 'a llist
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)

fixrec lconcat :: 'a llist llist \to 'a llist
where
lconcat \cdot lnil = lnil
| lconcat \cdot (lcons \cdot x \cdot xs) = lappend \cdot x \cdot (lconcat \cdot xs)

fixrec lmap :: ('a \to 'b) \to 'a llist \to 'b llist
where
lmap \cdot f \cdot lnil = lnil
| lmap \cdot f \cdot (lcons \cdot x \cdot xs) = lappend \cdot (lmap \cdot f \cdot 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 \to S
where
returnS = (\Lambda x. lcons \cdot x \cdot lnil)

definition bindS :: S \to (nat discr \to S) \to S
where
bindS = (\Lambda x g. lconcat \cdot (lmap \cdot g \cdot 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 \to S \to S
where
addS \equiv (\Lambda x y. bindS \cdot x \cdot (\Lambda xv. bindS \cdot y \cdot (\Lambda yv. returnS \cdot (xv + yv))))

definition disjS :: S \to S \to S
where
disjS \equiv lappend

definition failS :: S
where
failS \equiv 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.
definition evalS-body :: (expr discr → nat discr llist) → (expr discr → nat discr llist)
where evalS-body ≡ Λ r e. case undiscr e of
  const n ⇒ returnS · (Discr n)
  | add e1 e2 ⇒ addS · (r · (Discr e1)) · (r · (Discr e2))
  | disj e1 e2 ⇒ disjS · (r · (Discr e1)) · (r · (Discr e2))
  | fail ⇒ failS
abbreviation evalS :: expr discr → nat discr llist where evalS ≡ fix · 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 nat discr llist.

type-synonym Obs = nat discr llist
type-synonym Failure = Obs
type-synonym Success = nat discr → Failure → 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 unit → Obs.

The monad over the continuation type K is as follows:

definition returnK :: nat discr → K where
  returnK ≡ (Λ x. Λ s f. s · x · f)

definition bindK :: K → (nat discr → K) → K where
  bindK ≡ Λ x g. Λ s f. x · (Λ xv f'. g · xv · s · f') · f

Our extra constants are defined as follows:

definition addK :: K → K → K where
  addK ≡ (Λ x y. bindK · x · (Λ xv. bindK · y · (Λ yv. returnK · (xv + yv))))

definition disjK :: K → K → K where
  disjK ≡ (Λ g h. Λ s f. g · s · (h · s · f))

definition failK :: K where
  failK ≡ Λ s f. f

The continuation semantics is again straightforward:

definition evalK-body :: (expr discr → K) → (expr discr → K)
where

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

abbreviation \text{evalK} :: \text{expr discr} \to \text{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} :: \text{S} \to \text{K} where
\[
\begin{align*}
\text{SK} \cdot \lnil & = \text{failK} \\
\text{SK} \cdot (\text{lcons} \cdot x \cdot xs) & = (\Lambda \ s \ f. \ s \cdot x \cdot (\text{SK} \cdot xs \cdot s \cdot f))
\end{align*}
\]

definition \text{unwrap} :: (\text{expr discr} \to \text{nat discr llist}) \to (\text{expr discr} \to \text{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} :: \text{K} \to \text{S} where
\[
\text{KS} \equiv (\Lambda \ k. \ k \cdot \text{lcons} \cdot \lnil)
\]

definition \text{wrap} :: (\text{expr discr} \to \text{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 \text{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 un 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}:
\[
f \cdot x \nsubseteq g \cdot x \implies f \nsubseteq g
\]
by (auto simp: \text{cfun-below-iff})

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

proof –

\textbf{let } \ ?\text{witness} = \Lambda \ e. (\Lambda \ s \ f. \ \text{lnil} :: K)

\textbf{have} (unwrap oo wrap) \ ?\text{witness} \ (\text{Discr fail}) \ \bot \ (\text{lcons} \ \theta \ \text{lnil})

\textbf{by} (simp add: \text{failK-def} \ \text{wrap-def} \ \text{unwrap-def} \ \text{KS-def})

\textbf{hence} (unwrap oo wrap) \ ?\text{witness} \ \not\subseteq \ ?\text{witness}

\textbf{by} (fastforce intro!: \text{cfun-not-below})

\textbf{thus } \ ?\text{thesis} \ \text{by} (simp add: \text{cfun-not-below})

qed

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

\textbf{definition eval-work :: expr discr \to K where}

\text{eval-work} \equiv \text{fix}(\text{unwrap oo evalS-body oo wrap})

\textbf{definition eval-ww :: expr discr \to nat discr list where}

\text{eval-ww} \equiv \text{wrap} \cdot \text{eval-work}

\textbf{lemma} evalS = eval-ww

\textbf{unfolding} eval-ww-def eval-work-def

\textbf{using} worker\_wrapper\_id[OF wrap-unwrap-id]

\textbf{by} simp

We now show how the monadic operations correspond by showing that \texttt{SK}

witnesses a \textit{monad morphism} (Wadler 1992, §6). As required by Danvy et al.
(2001, Definition 2.1), the mapping needs to hold for our specific operations

in addition to the common monadic scaffolding.

\textbf{lemma} \texttt{SK-returnS-returnK}:

\texttt{SK} \cdot (\texttt{returnS} \cdot x) = \texttt{returnK} \cdot x

\textbf{by} (simp add: \texttt{returnS-def} \ \texttt{returnK-def} \ \texttt{failK-def})

\textbf{lemma} \texttt{SK-lappend-distrib}:

\texttt{SK} \cdot (\texttt{lappend} \cdot xs \cdot ys) \cdot s \cdot f = \texttt{SK} \cdot xs \cdot s \cdot (\texttt{SK} \cdot ys \cdot s \cdot f)

\textbf{by} (induct \ \texttt{xs}) (simp-all add: \texttt{failK-def})

\textbf{lemma} \texttt{SK-bindS-bindK}:

\texttt{SK} \cdot (\texttt{bindS} \cdot x \cdot g) = \texttt{bindK} \cdot (\texttt{SK} \cdot x) \cdot (\texttt{SK} \cdot oo \ g)

\textbf{by} (induct \ \texttt{x})

(simp-all add: \texttt{cfun-eq-iff}

\texttt{bindS-def} \ \texttt{bindK-def} \ \texttt{failK-def}

\texttt{SK-lappend-distrib})

\textbf{lemma} \texttt{SK-addS-distrib}:

\texttt{SK} \cdot (\texttt{addS} \cdot x \cdot y) = \texttt{addK} \cdot (\texttt{SK} \cdot x) \cdot (\texttt{SK} \cdot y)

\textbf{by} (clarsimp simp: \texttt{cfcomp1}

\texttt{addS-def} \ \texttt{addK-def} \ \texttt{failK-def}

\texttt{SK-bindS-bindK} \ \texttt{SK-returnS-returnK})

\textbf{lemma} \texttt{SK-disjS-disjK}:

33
\[ SK \cdot (\text{disj} \cdot xs \cdot ys) = \text{disj}K \cdot (SK \cdot xs) \cdot (SK \cdot ys) \]

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

**lemma** \(SK\cdot\text{fail}S = \text{fail}K\)

**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** (intro cfun-eqI)

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

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

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

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 \((\text{unwrap oo evalS-body oo wrap}) \cdot r \cdot e' \cdot s \cdot f\) = \((\text{evalK-body oo unwrap oo wrap}) \cdot r \cdot e' \cdot s \cdot f\)

by simp

**qed**

**theorem** 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 \(O(n^2)\) nub into an \(O(n \log n)\) one

Andy Gill’s solution, mechanised.
10.1 The \textit{nub} function.

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

\texttt{where}

\texttt{nub-nil = nil}

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

\texttt{lemma nub-strict[simp]: nub\(\bot = \bot\)}

\texttt{by fixrec-simp}

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

\texttt{where}

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

\texttt{| nub-body-f\((x @@ xs)\) = x @@ f\(\text{\texttt{lfilter}}\(\neg oo (\Lambda y. x =_{\text{B}} y)\) \text{\texttt{xs})}}

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

\texttt{by (rule cfun-eqI, subst nub-def, subst nub-bodyunfold, 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 llist}

\texttt{definition}

\texttt{SetEmpty :: NatSet where}

\texttt{SetEmpty \equiv \text{\texttt{nil}}}

\texttt{definition}

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

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

\texttt{definition}

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

\texttt{SetMem \equiv \text{\texttt{inmember}}\(\text{\texttt{bpred}}\(=\))}

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

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

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

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

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

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

AndyG’s new type.

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

\texttt{definition}

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

\texttt{nextR = (\Lambda r. \text{\texttt{case ldropWhile}}\(\Lambda x. \text{\texttt{SetMem}} \cdot x \cdot (\text{\texttt{exceptR}} \cdot r)) \cdot (\text{\texttt{resultR}} \cdot r) \text{\texttt{of}}}

35
\[
\begin{align*}
\text{filterR} &:: \mathtt{Nat} \rightarrow \mathtt{R} \rightarrow \mathtt{R} \\
\text{filterR} &\equiv (\Lambda v \ r. \ R \cdot (\text{resultR} \cdot r) \cdot (\text{SetInsert} \cdot v \cdot (\text{exceptR} \cdot r))) \\
\text{c2a} &:: \mathtt{Nat\ llist} \rightarrow \mathtt{R} \\
\text{c2a} &\equiv (\Lambda xs. \ R \cdot xs \cdot \text{SetEmpty}) \\
\text{a2c} &:: \mathtt{R} \rightarrow \mathtt{Nat\ llist} \\
\text{a2c} &\equiv (\Lambda r. \ lfilter \cdot (\Lambda v. \ neg \cdot (\text{SetMem} \cdot v \cdot (\text{exceptR} \cdot r))) \cdot (\text{resultR} \cdot r)) \\
\text{wrap} &:: (\mathtt{R} \rightarrow \mathtt{Nat\ llist}) \rightarrow \mathtt{Nat\ llist} \\
\text{wrap} &\equiv (\Lambda f \ xs. \ f \cdot (\text{c2a} \cdot xs)) \\
\text{unwrap} &:: (\mathtt{Nat\ llist} \rightarrow \mathtt{Nat\ llist}) \rightarrow \mathtt{R} \\
\text{unwrap} &\equiv (\Lambda f \ r. \ f \cdot (\text{a2c} \cdot r)) \\
\text{TR-deMorgan} &:: \neg \cdot (x \lor y) = (\neg \cdot x \land (\neg \cdot y)) \\
&\text{by (rule trE[where } p=x], \ simp-all) \\
\text{case-maybe-case:} \\
&\begin{cases}
\text{Nothing} &\Rightarrow f \\
Just \cdot (h \cdot x \cdot xs) &\Rightarrow g \cdot a \cdot b
\end{cases}
\end{align*}
\]
apply (cases L, simp-all)
apply (case-tac h-a-llist)
apply simp

proof —

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

proof —

have ?rhs = (case (case ldropWhile\cdot (\Lambda x. \text{SetMem}\cdot x\cdot (\text{exceptR}\cdot w))\cdot (\text{resultR}\cdot w) \text{ of } \\
  \text{lnil} \Rightarrow \text{Nothing} \\
  \mid x :@ xs \Rightarrow \text{Just}\cdot (x, R\cdot xs\cdot (\text{exceptR}\cdot w))) \text{ of } \text{Nothing} \Rightarrow f \mid \text{Just}\cdot (x, r) \Rightarrow g\cdot x\cdot (a2c\cdot r))
  \end{align*}

by (simp add: nextR-def)

also have \ldots = (case ldropWhile\cdot (\Lambda x. \text{SetMem}\cdot x\cdot (\text{exceptR}\cdot w))\cdot (\text{resultR}\cdot w) \text{ of } \\
  \text{lnil} \Rightarrow f \mid x :@ xs \Rightarrow g\cdot x\cdot (a2c\cdot (R\cdot xs\cdot (\text{exceptR}\cdot w))))

using case-maybe-case[where L=ldropWhile\cdot (\Lambda x. \text{SetMem}\cdot x\cdot (\text{exceptR}\cdot w))\cdot (\text{resultR}\cdot w)
\text{ and } f=f \text{ and } g=\Lambda x\cdot r. g\cdot x\cdot (a2c\cdot r) \text{ and } h=\Lambda x\cdot xs. (x, R\cdot xs\cdot (\text{exceptR}\cdot w))]

by simp

also have \ldots = ?lhs

apply (simp add: a2c-def)

apply (cases resultR\cdot w)

apply simp-all

apply (rule-tac p=\text{SetMem}\cdot a\cdot (\text{exceptR}\cdot w) \text{ in } \text{trE})

apply simp-all

apply (induct-tac list)

apply simp-all

apply (rule-tac p=\text{SetMem}\cdot aa\cdot (\text{exceptR}\cdot w) \text{ in } \text{trE})

apply simp-all

done

finally show ?lhs = ?rhs by simp

qed

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

using filter-filter\text{ where } p=\text{Tr}\cdot \text{neg oo} \ (\Lambda y. x =_B y) \text{ and } q=\Lambda v. \text{Tr}\cdot \text{neg} \cdot (\text{SetMem}\cdot v\cdot (\text{exceptR}\cdot r))]

unfolding a2c-def filterR-def

by (cases r, simp-all add: \text{SetMem}\cdot \text{SetInsert} \text{ TR-deMorgan})

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

definition
  \text{nub-body'} :: (\text{R} \rightarrow \text{Nat list}) \rightarrow \text{R} \rightarrow \text{Nat list} \text{ where }
  \text{nub-body'} \equiv \Lambda f\cdot r. \text{case } a2c\cdot r \text{ of } \text{lnil} \Rightarrow \text{lnil} \\
  \mid x :@ xs \Rightarrow x :@ f\cdot (\text{c2a}\cdot (lfilter\cdot (\text{neg oo} \ (\Lambda y. x =_B y)))\cdot xs))

lemma nub-body\cdot nub-body'\cdot eq: \text{unwrap oo nub-body oo wrap } = \text{nub-body'}

unfolding nub-body-def nub-body'\cdot def \text{unwrap-def wrap-def a2c-def c2a-def}
by (\(\text{rule cfun-eqI}\))+
  , \(\text{case-tac } l\text{filter}(\Lambda \ v. \ Tr.\neg (\text{SetMem} \cdot v \cdot (\text{exceptR} \cdot \text{xa}))))\cdot (\text{resultR} \cdot \text{xa})
  , \(\text{simp-all add: fix-const}\)

definition
\(\text{nub-body'} :: (R \to \text{Nat list}) \to R \to \text{Nat list where} \)
\(\text{nub-body'} \equiv \Lambda f \cdot r. \text{case nextR} \cdot r \text{ of Nothing} \Rightarrow \text{nil} \)
\(\text{Just} \cdot (x, xs) \Rightarrow x :@ f \cdot (\text{c2a} \cdot (\text{lfilter} \cdot (\text{neg oo} \cdot (\Lambda y. x = B y)))) \cdot (\text{resultR} \cdot \text{xa}))\)

lemma \(\text{nub-body'}-\text{nub-body''-eq} : \text{nub-body'} = \text{nub-body''}\)
proof (\(\text{rule cfun-eqI}\))+
  fix \(f \cdot r\) show \(\text{nub-body'} \cdot f \cdot r = \text{nub-body''} \cdot f \cdot r\)
  unfolding \(\text{nub-body'}-\text{def} \text{nub-body''-def}\)
  using case-a2c-case-caseR[\(\text{where f=lnil and g=\Lambda x xs, x :@ f \cdot (\text{c2a} \cdot (\text{lfilter} \cdot (\text{Tr.}\neg oo \cdot (\Lambda y. x = B y))) \cdot xs)\) and w=r]\)
  by simp
qed

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

lemma \(\text{nub-body''-nub-body''''-eq} : \text{nub-body''} = \text{nub-body'''} \text{ oo (unwrap oo wrap)}\)
unfolding \(\text{nub-body''-def} \text{nub-body''''-def} \text{wrap-def unwrap-def}\)
by (\(\text{rule cfun-eqI}\))+, \(\text{simp add: filter-filterR}\)

Finally glue it all together.

lemma \(\text{wrap-nub-body''''-eq} : \text{nub} = \text{wrap} \cdot (\text{fix-nub-body''''})\)
using worker-wrapper-fusion-new[\(\text{OF wrap-unwrap-id unwrap-strict, where body=nub-body}\]
\(\text{nub-nub-body-eq}\)
\(\text{nub-body-nub-body'-eq}\)
\(\text{nub-body'-nub-body''-eq}\)
\(\text{nub-body''-nub-body''''-eq}\)
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 \to 'a
where
\(\text{llast} \cdot (x :@ yys) = (\text{case yys of \text{nil} \Rightarrow x | y :@ ys \Rightarrow \text{llast} \cdot yys})\)
lemma llast-strict[simp]: llast · ⊥ = ⊥
by fixrec-simp

fixrec llast-body :: ('a list → 'a) → 'a list → '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 list → 'a)
where
wrap ≡ Λ f·(x :@ xs). f·x·xs

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

lemma unwrap-strict[simp]: unwrap·⊥ = ⊥
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 list → 'a) → 'a → 'a list → 'a where
llast-worker ≡ Λ r x yys. case yys of lnil ⇒ x | y :@ ys ⇒ r·y·ys

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

lemma llast-worker-llast-body: llast-worker = unwrap oo llast-body oo wrap
unfolding llast-body-def wrap-def 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 assms = ?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-wraper-body[OF wrap-unwrap-ID])
also have ... = wrap·(fix·llast-worker)
  by (simp only: llast-worker-llast-body)
also have ... = ?rhs 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 \( f :: \text{Nat} \to a \) is not strict – in fact constant, as \( \text{Nat} \) is a flat domain – then \( f \perp \neq \perp = (\text{map } f \ [0..]) \ !! \perp \), where \( xs !! n \) is the \( n \)th element of \( xs \).

References


