Mechanising the worker/wrapper transformation

Peter Gammie

May 28, 2015

Contents

1 Introduction 2

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

3 The transformation according to Gill and Hutton 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
10 Transforming $O(n^2)$ \textit{nub} into an $O(n \log n)$ one \hfill 34
  10.1 The \textit{nub} function. \hfill 34
  10.2 Optimised data type. \hfill 35

11 Optimise “last”. \hfill 38
  11.1 The \textit{last} function. \hfill 38

12 Concluding remarks \hfill 39

Bibliography \hfill 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, $\lambda\_\_\_\_$ for continuous function abstraction, $\_\_\_\_$ for continuous function application, \texttt{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).

\textbf{lemma} rolling-rule-ltr: \texttt{fix}-(g oo f) $\sqsubseteq$ g-(\texttt{fix}-(f oo g))

\textbf{proof} –
  \texttt{have} g-(\texttt{fix}-(f oo g)) $\sqsubseteq$ g-(\texttt{fix}-(f oo g))
  \texttt{by (rule below-refl) — reflexivity}
  \texttt{hence} g-(((f oo g)-(\texttt{fix}-(f oo g)))) $\sqsubseteq$ g-(\texttt{fix}-(f oo g))
  \texttt{using} fix-eq[\texttt{where} \texttt{F}=f oo g] \texttt{by simp — computation}
  \texttt{hence} (g oo f)-(g-(\texttt{fix}-(f oo g)))) $\sqsubseteq$ g-(\texttt{fix}-(f oo g))
  \texttt{by simp — re-associate op oo}
  \texttt{thus} \texttt{fix}-(g oo f) $\sqsubseteq$ g-(\texttt{fix}-(f oo g))
using \textit{fix-least-below} by blast — induction

\textbf{Qed}

\textbf{Lemma} \textit{rolling-rule-rtl}:
\[ g \cdot (\text{fix} \cdot (f \circ o g)) \sqsubseteq \text{fix} \cdot (g \circ o f) \]
\textbf{Proof} —
\begin{itemize}
  \item have \text{fix} \cdot (f \circ o g) \sqsubseteq f \cdot (\text{fix} \cdot (g \circ o f)) \text{ by (rule rolling-rule-ltr)}
  \item hence \text{fix} \cdot (f \circ o g) \sqsubseteq g \cdot (f \cdot (\text{fix} \cdot (g \circ o f)))
  \item by (rule monofun-cfun-arg) — g is monotonic
  \item thus \text{fix} \cdot (f \circ o g) \sqsubseteq \text{fix} \cdot (g \circ o f)
  \item using \textit{fix-eq[where } F = g \circ o f]\text{ by simp — computation}
\end{itemize}
\textbf{Qed}

\textbf{Lemma} \textit{rolling-rule}:
\[ \text{fix} \cdot (g \circ o f) = g \cdot (\text{fix} \cdot (f \circ o g)) \]
\textbf{by (rule below-antisym[OF rolling-rule-ltr 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 \( g \):

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 \textit{parallel-fix-ind}:

\[ \begin{align*}
  \text{adm} & \quad (\lambda x. ?P \ (f s t \ x) \ (s n d \ x)) \\
  \begin{array}{c}
  ?P \perp \perp \\
  \bigwedge x y \quad ?P \ x y \\
  \frac{?P \ (\forall F \cdot x) \ (\forall G \cdot y)}{?P \ (\text{fix} \cdot ?F) \ (\text{fix} \cdot ?G)}
  \end{array}
\end{align*} \]

in a very straightforward way:

\textbf{Lemma} \textit{lfp-fusion}:

\begin{itemize}
  \item \textbf{Assumes} \( g \perp = \perp \)
  \item \textbf{Assumes} \( g \circ o f = h \circ o g \)
  \item \textbf{Shows} \( \text{g} \cdot (\text{fix} \cdot f) = \text{fix} \cdot h \)
\end{itemize}
\textbf{Proof} (\textit{induct rule: parallel-fix-ind})
For a recursive definition \( \text{comp} = \text{fix} \cdot \text{body} \) 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:

\[
\text{comp} = \text{wrap} \cdot \text{work}
\]

\[
\text{work} :: B
\]

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

(the worker/wrapper transformation)

Also:

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

(worker/wrapper fusion)

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

```isar
case 2 show \( g \cdot \bot = \bot \) by fact
  case (3 x y)
  from \( g \cdot x = y \) : \( g \circ \circ f = h \circ \circ g \)
  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 \( \text{wrap} \circ \circ \text{unwrap} \) to be the identity for all values.

```isar
lemma worker-wrapper-id:
  fixes \( \text{wrap} :: 'b::pcpo \rightarrow 'a::pcpo \)
  fixes \( \text{unwrap} :: 'a \rightarrow 'b \)
  assumes \( \text{wrap-unwrap}: \text{wrap} \circ \circ \text{unwrap} = \text{id} \)
  assumes \( \text{comp-body}: \text{computation} = \text{fix} \cdot \text{body} \)
  shows \( \text{computation} = \text{wrap} \cdot (\text{fix} \cdot (\text{unwrap} \circ \circ \text{body} \circ \circ \text{wrap})) \)
proof −
  from \( \text{comp-body} \) have \( \text{computation} = \text{fix} \cdot (\text{id} \circ \circ \text{body}) \)
```
by simp
also from wrap-unwrap have \ldots = \text{fix-}(\text{wrap oo unwrap oo body})
by (simp add: assoc-oo)
also have \ldots = \text{wrap-}(\text{fix-}(\text{unwrap oo body oo wrap}))
using rolling-rule[where \(f=\text{unwrap oo body}\) and \(g=\text{wrap}\)]
by (simp add: assoc-oo)
finally show ?thesis .
qed

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

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

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

**Lemma** worker-wrapper-fusion:

- **Fixes** \( \text{wrap} : 'b :: \text{pcpo} \rightarrow 'a :: \text{pcpo} \)
- **Fixes** \( \text{unwrap} : 'a \rightarrow 'b \)
- **Assumes** \( \text{wrap-unwrap}: \text{wrap} \circ \text{unwrap} = \text{ID} \)
- **Assumes** \( \text{work}: \text{work} = \text{fix}(\text{unwrap} \circ \text{body} \circ \text{wrap}) \)
- **Shows** \( (\text{unwrap} \circ \text{wrap}) \circ \text{work} = \text{work} \)

**Proof** –

- Have \( (\text{unwrap} \circ \text{wrap}) \circ \text{work} = (\text{unwrap} \circ \text{wrap}) \circ (\text{fix}(\text{unwrap} \circ \text{body} \circ \text{wrap})) \)
  - Using **work by simp**
- Also have \( \ldots \circ (\text{unwrap} \circ \text{work}) = (\text{fix}(\text{unwrap} \circ \text{body} \circ \text{work} \circ \text{unwrap} \circ \text{work}) \)
  - Using **wrap-unwrap by (simp add: assoc-oo)**
- Also have \( \ldots = \text{fix}(\text{unwrap} \circ \text{work} \circ \text{unwrap} \circ \text{body} \circ \text{work}) \)
  - Using **rolling-rule [where \( f = \text{unwrap} \circ \text{body} \circ \text{work} \) and \( g = \text{unwrap} \circ \text{work} \)] by (simp add: assoc-oo)**
- Also have \( \ldots = \text{fix}(\text{unwrap} \circ \text{body} \circ \text{work}) \)
  - Using **wrap-unwrap by (simp add: assoc-oo)**
- Finally show ?thesis using **work by simp**

qed

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

### 3.1 Worker/wrapper fusion is partially correct

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

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

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

\[
\frac{f \cdot ?x = ?x}{\text{fix}_x.f \subseteq ?x} \quad \text{fix}_x \text{least}
\]
viz that \( f \ x = x \) 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 \( \text{unwrap} \ oo \ \text{body} \ oo \ \text{wrap} \), can be rewritten as \( \text{body}' \ oo \ \text{unwrap} \ oo \ \text{wrap} \). 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:

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

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

**lemma** \( \text{fusion-partially-correct} \):

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

**proof** (rule \( \text{fix-least} \))

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

**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** \( A = A \)

---

7
domain $B = B$ (lazy $A$)

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

```plaintext
fixrec wrap :: $B \to A$
where wrap $(B \cdot a) = a$

fixrec unwrap :: $A \to B$
where unwrap = $B$
```

Discharging the worker/wrapper hypothesis is similarly routine.

```plaintext
lemma wrap-unwrap:: wrap oo unwrap = ID
  by (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.

```plaintext
fixrec body :: $A \to A$
where body $r = A$
```

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

```plaintext
fixrec body' :: $B \to B$
where body' $(B \cdot a) = B \cdot A$
```

As explained above, we set up the fusion opportunity:

```plaintext
lemma body-body': unwrap oo body oo wrap = body' oo unwrap oo wrap
  by (simp add: cfun-eq-iff)
```

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

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

```plaintext
lemma fix-body' $\subseteq$ fix-(unwrap oo body oo wrap)
  by (rule fusion-partially-correct[OF wrap-unwrap refl body-body'])
```

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

```plaintext
lemma ~ 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
```

This trick works whenever `unwrap` is not strict. In the following section we show that requiring `unwrap` to be strict leads to a straightforward proof of total correctness.
Note that if we have already established that \( \text{wrap} \circ \text{unwrap} = \text{ID} \), then making \( \text{unwrap} \) strict preserves this equation:

\[
\begin{align*}
\text{lemma} & \quad \text{assumes } \text{wrap} \circ \text{unwrap} = \text{ID} \\
& \quad \text{shows } \text{wrap} \circ \text{strictify} \cdot \text{unwrap} = \text{ID} \\
\text{proof} & \quad (\text{rule cfun-eqI}) \\
& \quad \text{fix } x \\
& \quad \text{from } \text{assms} \\
& \quad \text{show } (\text{wrap} \circ \text{strictify} \cdot \text{unwrap}) \cdot x = \text{ID} \cdot x \\
& \quad \quad \text{by } (\text{cases } x = \bot) \ (\text{simp-all add: cfun-eq-iff retraction-strict})
\end{align*}
\]

qed

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

4 A totally-correct fusion rule

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

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

\[
\begin{align*}
\text{lemma } \text{worker-\text{wrapp}\text{er-fusion-new}:} & \\
& \quad \text{fixes } \text{wrap} :: 'b::\text{pcpo} \rightarrow 'a::\text{pcpo} \\
& \quad \text{fixes } \text{unwrap} :: 'a \rightarrow 'b \\
& \quad \text{fixes } \text{body}' :: 'b \rightarrow 'b \\
& \quad \text{assumes } \text{wrap-unwrap: } \text{wrap} \circ \text{unwrap} = (\text{ID} :: 'a \rightarrow 'a) \\
& \quad \text{assumes } \text{unwrap-strict: } \text{unwrap} \cdot \bot = \bot \\
& \quad \text{assumes } \text{body-body': } \text{unwrap} \circ \text{body} \circ \text{unwrap} = \text{body'} \circ (\text{unwrap} \circ \text{unwrap}) \\
& \quad \text{shows } \text{fix-body} = \text{wrap-(fix-body')} \\
\text{proof} & \quad (\text{from } \text{body-body'}) \\
& \quad \quad \text{have } \text{unwrap} \circ \text{body} \circ (\text{unwrap} \circ \text{unwrap}) = (\text{body'} \circ \text{unwrap} \circ (\text{unwrap} \circ \text{unwrap})) \\
& \quad \quad \quad \quad \text{by } (\text{simp add: assoc-oo}) \\
& \quad \quad \text{with } \text{wrap-unwrap} \text{ have } \text{unwrap} \circ \text{body} = \text{body'} \circ \text{unwrap} \\
& \quad \quad \quad \quad \text{by } \text{simp} \\
& \quad \quad \text{with } \text{unwrap-strict} \text{ have } \text{unwrap} \cdot (\text{fix-body}) = \text{fix-body'} \\
& \quad \quad \quad \quad \text{by } (\text{rule lfp-fusion}) \\
& \quad \quad \text{hence } (\text{wrap} \circ \text{unwrap}) \cdot (\text{fix-body}) = \text{wrap} \cdot (\text{fix-body'}) \\
& \quad \quad \quad \quad \text{by } \text{simp} \\
& \quad \quad \text{with } \text{wrap-unwrap} \text{ show } \text{thesis by } \text{simp} \\
\text{qed}
\end{align*}
\]
We can also show a more general result which allows fusion to be optionally performed on a per-recursive-call basis using parallel_fix_ind:

**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 = \(\lambda(x, y). x = y \land \text{unwrap}·\text{wrap}·x = x\)

have ?P (fix·(unwrap oo body oo wrap), (fix·body'))

**proof**(induct rule: parallel-fix-ind)

- **case 2** with retraction-strict unwrap-strict wrap-unwrap show ?P (⊥, ⊥)
  - by (bestsimp simp add: cfun-eq-iff)
- **case** (3 x y)
  - **hence** xy: x = y and unwrap-wrap: unwrap·(wrap·x) = x by auto
  - from body-body' xy unwrap-wrap
  - have (unwrap oo body oo wrap)·x = body'·y
    - by simp
  - moreover
  - from wrap-unwrap
  - have unwrap·(wrap·(unwrap oo body oo wrap)·x)) = (unwrap oo body oo wrap)·x
    - by (simp add: cfun-eq-iff)
  - **ultimately show** ?case by simp

**qed** simp

**thus** ?thesis

- using worker-wrapper-id[OF wrap-unwrap refl] by simp

**qed**

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

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

5 Naive reverse becomes accumulator-reverse.

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

The “Hughes” list type.

**type-synonym** 'a H = 'a list → 'a list

**definition**
For a recursive definition \( \text{comp} = \text{body} \) of type \( A \) and a pair of functions \( \text{wrap} :: B \to A \) and \( \text{unwrap} :: A \to B \) where \( \text{wrap} \circ \text{unwrap} = \text{id}_A \) and \( \text{unwrap} \perp = \perp \), define:

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

(the worker/wrapper transformation)

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

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

(worker/wrapper fusion)

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

\[
\text{list2H} :: \mathcal{A}\text{llist} \to \mathcal{A}\text{H}
\]

\[
\text{list2H} \equiv \text{lappend}
\]

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

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

**Definition**

\[
\text{H2list} :: \mathcal{A}\text{H} \to \mathcal{A}\text{llist}
\]

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

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** \( \text{H-list-hom-append}: \text{list2H} \cdot (xs :++ ys) = \text{list2H} \cdot xs \oo \text{list2H} \cdot ys \) (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 \( \ldots = (\text{list2H} \cdot xs \oo \text{list2H} \cdot ys) \cdot zs \) by simp

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

qed

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

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

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

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

**fixrec** \( \text{lrev} :: \mathcal{A}\text{llist} \to \mathcal{A}\text{llist} \)
where
\[ \text{lrev·nil} = \text{nil} \]
\[ \text{lrev·(x :@ xs)} = \text{lrev·xs :++ (x :@ nil)} \]

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

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

**fixrec** \text{lrev-body :: \(\forall \ a \ llist \rightarrow \ llist\)}

\[ \text{lrev-body·r·nil} = \text{nil} \]
\[ \text{lrev-body·r·(x :@ xs)} = r·xs :++ (x :@ \text{nil}) \]

**lemma** \text{lrev-body-strict [simp]}: \text{lrev-body·r·⊥ = ⊥}
**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 = fix·lrev-body}
**by** (rule \text{cfun-eqI}, subst \text{lrev-def}, subst \text{lrev-body.unfold}, simp)

Wrap / unwrap functions.

**definition**
\[ \text{unwrapH :: \(\forall \ a \ llist \rightarrow \ llist\)} \]
\[ \text{unwrapH ≡ \Lambda \ f \ xs . list2H·(f·xs)} \]

**lemma** \text{unwrapH-strict [simp]}: \text{unwrapH·⊥ = ⊥}
**by** \text{unfolding \text{unwrapH-def} by (rule \text{cfun-eqI}, simp)}

**definition**
\[ \text{wrapH :: \(\forall \ a \ llist \rightarrow \ llist\)} \]
\[ \text{wrapH ≡ \Lambda \ f \ xs . H2list·(f·xs)} \]

**lemma** \text{wrapH-unwrapH-id; wrapH oo unwrapH = ID (is ?lhs = ?rhs)}
**proof** (rule \text{cfun-eqI}):
\[ \text{fix f xs} \]
\[ \text{have ?lhs·f·xs = H2list·(list2H·(f·xs)) by (simp add: wrapH-def unwrapH-def)} \]
\[ \text{also have \ldots = (H2list oo list2H)·(f·xs) by simp} \]
\[ \text{also have \ldots = ID·(f·xs) by (simp only: H2list-list2H-inv)} \]
\[ \text{also have \ldots = ?rhs·f·xs by simp} \]
\[ \text{finally show ?lhs·f·xs = ?rhs·f·xs}. \]
\[ \text{qed} \]

5.2 Gill/Hutton-style worker/wrapper.

**definition**
\[ \text{lrev-work :: \(\forall \ a \ llist \rightarrow \ llist\)} \]
\[ \text{lrev-work ≡ fix·(unwrapH oo lrev-body oo wrapH)} \]
definition
lrev-wrap :: 'a llist → 'a llist where
lrev-wrap ≡ wrapH·lrev-work

lemma lrev-lrev-ww-eq: lrev = lrev-wrap
using worker-wwrapperv-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 (subst 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·⊥ = ⊥
unfoldin lrev-work2-def
by (subst fix-eq) simp
lemma \texttt{lrev-body2-lrev-body1-eq}: \texttt{lrev-body2} = \texttt{lrev-body1}
by ((\texttt{rule cfun-eqI})+, (\texttt{subst \ lrev-body1.unfold, subst \ lrev-body2.unfold})
\texttt{, (simp add: H-list-hom-append\[\texttt{symmetric}\] H-list-hom-id)})

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

Simplify.

\texttt{fixrec \ lrev-body3 :: ('a \texttt{llist} \to 'a \texttt{H} \to 'a \texttt{llist} \to 'a \texttt{H}}
\texttt{where}
\texttt{lrev-body3} · \texttt{r} · \texttt{lnil} = \texttt{ID}
\texttt{| lrev-body3} · \texttt{r} · (\texttt{x} :\@ \texttt{x} \texttt{s}) = \texttt{r} · \texttt{x} oo \texttt{list2H} · (\texttt{x} :\@ \texttt{lnil})

lemma \texttt{lrev-body3-strict[simp]}: \texttt{lrev-body3} · \texttt{⊥} = \texttt{⊥}
by \texttt{fixrec-simp}

definition \texttt{lrev-work3} :: 'a \texttt{llist} \to 'a \texttt{H}
\texttt{where}
\texttt{lrev-work3} \equiv \texttt{fix} · \texttt{lrev-body3}

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

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

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

\textbf{lemma} \texttt{lrev-work3-lrev-work2-eq}: \texttt{lrev-work3 = lrev-work2}  \textit{(is \ ?lhs = \ ?rhs)}
\begin{proof}
\textit{rule cfun-eqI}
\end{proof}

\textbf{next}
\begin{verbatim}
case \textit{lnil} \textbf{thus} \textit{?thesis}
  unfolding \texttt{lrev-work2-def}
  by (subst \textit{fix-eq[where F=lrev-body2], simp})
\end{verbatim}
\begin{verbatim}
next
case (lcons \textit{y} \textit{ys})
  hence \texttt{lrev-body3-lrev-work2-ys = lrev-work2-ys oo list2H-\{y :@ \textit{lnil}\}} \textbf{by} \textit{simp}
  also have \ldots = list2H-\{(wrapH-lrev-work2)-ys\} oo list2H-\{y :@ \textit{lnil}\}
  using \texttt{lrev-wafusion[where xs=ys]} \textbf{by} \textit{simp}
  also from \textit{lcons} have \ldots = \texttt{lrev-body2-lrev-work2-ys} \textbf{by} \textit{simp}
  also have \ldots = \texttt{lrev-work2-\{x\}}
  unfolding \texttt{lrev-work2-def} \textbf{by} \textit{(simp only: fix-eq[symmetric])}
  finally show \textit{?thesis} \textbf{by} \textit{simp}
\end{verbatim}
\textbf{ qed}
\end{verbatim}
\begin{verbatim}
{ thus \texttt{lrev-body3-lrev-work2 = lrev-work2} \textbf{by} \textit{(rule cfun-eqI)}
\textbf{ qed}
\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 (subst lrev-body2.unfold, subst lrev-body3.unfold)
apply (rule cfun-eqI)+
apply (case-tac xa)
  apply (simp-all add: unwrapH-def)
done

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

Final syntactic tidy-up.

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

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

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

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

lemma lrev-body-final-lrev-body3-eq: lrev-body-final = lrev-body3
by (simp only: lrev-body-final-lrev-body3-eq' cfun-eqI)
lemma \texttt{lrev-final-lrev-eq}: \texttt{lrev} = \texttt{lrev-final} (\texttt{is} \ ?\texttt{lhs} = \ ?\texttt{rhs})

proof –
  have \ ?\texttt{lhs} = \texttt{lrev-wrap} by (rule \texttt{lrev-lrev-ww-eq})
  also have \ldots = \texttt{wrapH}·\texttt{lrev-work} by (simp only: \texttt{lrev-wrap-def})
  also have \ldots = \texttt{wrapH}·\texttt{lrev-work1} by (simp only: \texttt{lrev-work1-lrev-work-eq})
  also have \ldots = \texttt{wrapH}·\texttt{lrev-work2} by (simp only: \texttt{lrev-work2-lrev-work1-eq})
  also have \ldots = \texttt{wrapH}·\texttt{lrev-work3} by (simp only: \texttt{lrev-work3-lrev-work2-eq})
  also have \ldots = \texttt{lrev-final} by (simp add: \texttt{lrev-final-def} \texttt{cfun-eqI} \texttt{H2list-def} \texttt{wrapH-def})
finally show \ ?\texttt{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 \texttt{UNat} denotes the discrete domain of naturals, \texttt{UNat\_⊥} the lifted (flat and pointed) variant, and \texttt{Nat} the standard boxed domain, isomorphic to \texttt{UNat\_⊥}. This latter distinction helps us keep the boxed naturals and lifted function codomains separated; applications of \texttt{unbox} should be thought of in the same way as Haskell’s \texttt{newtype} constructors, i.e. operationally equivalent to \texttt{ID}. The divergence monad is used to handle the unboxing, see below.

6.1 Factorial example.

Standard definition of factorial.

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

unfolding \textit{fac-body-def} \textit{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 \to Nat) \to UNat \to UNat \bot where
unwrapB \equiv \Lambda f. unbox oo f oo box

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

definition
wrapB :: (UNat \to UNat \bot) \to Nat \to Nat where
wrapB \equiv \Lambda f x . unbox.x >>=} f >>=} box

lemma \textit{wrapB-unwrapB-body}:
assumes \textit{strictF}: f \bot = \bot
shows (wrapB oo unwrapB).f = f (is ?\textit{lhs} = ?\textit{rhs})
proof (rule cfun-eqI)
fix x :: Nat
have ?\textit{lhs}.x = unbox.x >>=} (\Lambda x'. unwrapB.f.x' >>=} box)
unfolding wrapB-def \textit{by simp}
also have \ldots = unbox.x >>=} (\Lambda x'. unbox.(f.box.x')) >>=} box)
unfolding unwrapB-def \textit{by simp}
also from \textit{strictF} have \ldots = f.x by (cases x, simp-all)
finally show ?\textit{lhs}.x = ?\textit{rhs}.x .
qed

Apply worker/wrapper.

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

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

lemma \textit{fac-fac-ww-eq}: fac = fix fac-wrap (is ?\textit{lhs} = ?\textit{rhs})

proof
have \textit{wrapB} oo \textit{unwrapB} oo \textit{fac-body} = \textit{fac-body}
using \textit{wrapB-unwrapB-body}[OF \textit{fac-body-strict]}
by \textit{(rule cfun-eqI, simp)}
thus ?\textit{thesis}
using worker-wrapper-body[where \textit{computation}=fac \textit{and body}=\textit{fac-body} \textit{and wrap}=\textit{wrapB} \textit{and unwrap}=\textit{unwrapB}]
This is not entirely faithful to the paper, as they don’t explicitly handle the lifting of the codomain.

**definition**

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

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

**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-work-def)
  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 op >>>, 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
\[ i\text{-th}(x \&\& xs) = \text{Nat-case}\cdot x\cdot(i\text{-th}\cdot xs) \]

**abbreviation**

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

\[ s \ !! i \equiv i\text{-th}\cdot s\cdot i \]

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**

\[ \text{unwrapS}' :: (Nat \rightarrow \text{a}) \rightarrow \text{a} \text{ Stream} \text{ where} \]

\[ \text{unwrapS}' \equiv \Lambda f. \text{smap}\cdot f\cdot \text{nats} \]

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

**fixrec** `unwrapS :: (Nat \rightarrow \text{a}) \rightarrow \text{a} \text{ Stream}`

**where**

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

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

**lemma** `unwrapS-unwrapS'-eq`: `unwrapS = unwrapS' (is ?lhs = ?rhs)`

**proof**

1. **fix** `f`
2. **show** ?lhs\cdot f = ?rhs\cdot f

**proof**

1. **(coinduct rule: Stream.coinduct)**
2. **let** ?R = \( \lambda s s'. (\exists f. s = f\cdot 0 \&\& \text{unwrapS}\cdot(f oo (\Lambda x. 1 + x)) \wedge s' = f\cdot 0 \&\& \text{smap}\cdot(f oo (\Lambda x. 1 + x))\cdot \text{nats}) \)
3. **show** Stream-bisim ?R

**proof**

1. **fix** `s s'`
2. **assume** ?R \ s \ s'
3. **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 \text{nats}`

**by** blast

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

**by** ( **rule** exI[where `x=foo (\Lambda x. 1 + x)`]

1. `\text{smap}\cdot(\text{unwrapS}\cdot(\text{unwrapS}\cdot(\text{unwrapS}\cdot(\text{unwrapS}\cdot(\text{unwrapS}\cdot(\text{unwrapS}\cdot(f oo (\Lambda x. 1 + x))))))))`)

**with** `fs fs'`

5. **show** `(s = \bot \& s' = \bot) \lor (\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 \text{nats}) \wedge s = h \&\& t \wedge s' = h \&\& t')` **by** best

**qed**

23
show ?R (?lhs·f) (?rhs·f)

proof –
  have lhs: ?lhs·f = f·0 &\& unwrapS·(f oo (\Lambda x. 1 + x)) by (subst unwrapS.unfold, simp)
  have rhs: ?rhs·f = f·0 &\& smap·(f oo (\Lambda x. 1 + x))·nats by (rule unwrapS'-unfold)
  from lhs rhs show ?thesis by best
qed

definition
wrapS :: 'a Stream \to Nat \to 'a where
wrapS ≡ \Lambda s i . s !! i

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

In practice all functions worth memoising are strict in the memoised argu-
ment.

lemma wrapS-unwrapS-id':
  assumes strictF: (f::Nat \to 'a)·⊥ = ⊥
  shows unwrapS·f !! n = f·n
using strictF
proof((induct n arbitrary; f rule: Nat-induct)
  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 (\Lambda x. 1 + x))) !! (i + 1)
  by (subst unwrapS.unfold, simp)
also from Suc have \ldots = unwrapS·(f oo (\Lambda x. 1 + x)) !! i by simp
also from Suc have \ldots = (f oo (\Lambda 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
fib-body :: (Nat \to Nat) \to Nat \to Nat where
fib-body ≡ \Lambda r. Nat-case·1·(Nat-case·1·(\Lambda n. r·n + r·(n + 1)))

definition
fib :: Nat \to Nat where
fib ≡ fix·fib-body
Apply worker/wrapper.

definition

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

definition

fib-wrap :: Nat → Nat where
fib-wrap ≡ wrapS ∘ fib-work

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

lemma fib-fw-eq: fib = fib-wrap
using worker-unwrap-body[OF unwrapS-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 ∘ 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 Bekič’s Theorem (Bekič 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′ + (n + 1))))
let ?mr = Λ (fif : Nat Stream, fff). (smap · fff ∘ nats, ?wb ∘ fif)
have ?lhs = fst (fix ∘ ?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 \( \ldots = (\mu \text{fuf}. \text{fst} (\text{?mr} \cdot (\text{fuf}, \mu \text{fff} \cdot \text{snd} (\text{?mr} \cdot (\text{fuf}, \text{fff})))))) \)
using \text{fix-cprod}[\text{where } F = \text{?mr}] \text{ by simp}
also have \( \ldots = (\mu \text{fuf}. \text{smap} (\mu \text{fff} \cdot \text{?wb-fuf} \cdot \text{nats} ) \text{ by simp}) \)
also have \( \ldots = (\mu \text{fuf}. \text{smap} (\text{?wb-fuf} \cdot \text{nats} ) \text{ by simp add: fix-const}) \)
also have \( \ldots = \text{?rhs} \)
unfolding \text{fib-body-def} \text{fib-def} \text{unwrapS-def} \text{unwrapS'}-def \text{wrapS-def} \text{ 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 \( \text{?lhs} = (\Lambda n. \text{fib-work-final} !! n) \text{ by simp add: fib-final-def} \)
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 \( \ldots = \text{?rhs} \text{ by simp only: fib-ww-eq} \)
finally show \( \text{?thesis} \).
qed

8 Tagless interpreter via double-barreled continuations

type-synonym \( 'a \text{ Cont} = ( 'a \rightarrow 'a) \rightarrow 'a \)
definition \text{val2cont} :: \( 'a \rightarrow 'a \text{ Cont} \text{ where} \)
\text{val2cont} \equiv (\Lambda a c. c \cdot a)
definition \text{cont2val} :: \( 'a \text{ Cont} \rightarrow 'a \text{ where} \)
\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: val2cont-def cont2val-def)
domain \text{Expr} =
  \text{Val} (\text{lazy val::Nat})
  | \text{Add} (\text{lazy addl::Expr}) (\text{lazy addr::Expr})
  | \text{Throw}
  | \text{Catch} (\text{lazy cbody::Expr}) (\text{lazy Chandler::Expr})
fixrec \text{eval} :: \text{Expr} \rightarrow \text{Nat} \text{ Maybe} \text{ where} \)
\text{eval}.(\text{Val} \cdot n) = \text{Just} \cdot n
| \text{eval}.(\text{Add} :: x \cdot y) = \text{mliftM2} (\Lambda a b. a + b) \cdot (\text{eval} \cdot x) \cdot (\text{eval} \cdot y)
| \text{eval}.\text{Throw} = \text{mfail}
| \text{eval}. (\text{Catch} :: x \cdot y) = \text{mcatch} \cdot (\text{eval} \cdot x) \cdot (\text{eval} \cdot y)
fixrec eval-body :: (Expr → Nat Maybe) → Expr → Nat Maybe
where
  eval-body r (Val n) = Just n
| eval-body r (Add x y) = mleftM2 (Λ a b. a + b) (r x) (r y)
| eval-body r Throw = mfail
| eval-body r (Catch x y) = mcatch (r x) (r y)

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

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

8.1 Worker/wrapper

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

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

definition wrapC :: (Expr → (Nat → Nat Maybe) → Nat Maybe) → (Expr
  → Nat Maybe) where
  wrapC ≡ Λ g e. g · e · Just · Nothing

lemma wrapC-unwrapC-id: wrapC oo unwrapC = ID
proof (intro cfun-eqI)
  fix g e
  show (wrapC oo unwrapC) · g · e = ID · g · e
    by (cases g · e, simp-all add: wrapC-def unwrapC-def)
qed

definition eval-work :: Expr → (Nat → Nat Maybe) → Nat Maybe
  → Nat Maybe where
  eval-work ≡ fix (unwrapC oo eval-body oo wrapC)

definition eval-wrap :: Expr → Nat Maybe where
  eval-wrap ≡ wrapC · eval-work

fixrec eval-body’ :: (Expr → (Nat → Nat Maybe) → Nat Maybe)
  → Expr → (Nat → Nat Maybe) → Nat Maybe where
  eval-body’ r · (Val n) · s · f = s · n
| eval-body’ r · (Add x y) · s · f = (case wrapC · r · x of
\[ \text{eval-body}' \cdot \text{r} \cdot \text{Throw} \cdot \text{s} \cdot f = f \]
\[ \text{eval-body}' \cdot \text{r} \cdot (\text{Catch} \cdot x \cdot y) \cdot \text{s} \cdot f = (\text{case} \ \text{wrapC} \cdot \text{r} \cdot x \ of \]
\[ \text{Nothing} \rightarrow (\text{case} \ \text{wrapC} \cdot \text{r} \cdot y \ of \]
\[ \text{Nothing} \rightarrow f \]
\[ \text{Just} \cdot n \rightarrow s \cdot (n + m)) \]
\[ \text{eval-body-final} : (\text{Expr} \rightarrow (\text{Nat} \rightarrow \text{Nat Maybe}) \rightarrow \text{Nat Maybe} \rightarrow \text{Nat Maybe}) \rightarrow \text{Expr} \rightarrow (\text{Nat} \rightarrow \text{Nat Maybe}) \rightarrow \text{Nat Maybe} \rightarrow \text{Nat Maybe} \]
\[ \text{eval-body-final} \cdot \text{r} \cdot \text{Val} \cdot n \cdot \text{s} \cdot f = \text{s} \cdot n \]
\[ \text{eval-body-final} \cdot \text{r} \cdot (\text{Add} \cdot x \cdot y) \cdot \text{s} \cdot f = \text{r} \cdot x \cdot (\Lambda \ n, \text{r} \cdot y \cdot (\Lambda \ m, \text{s} \cdot (n + m)) \cdot f) \cdot f \]
\[ \text{eval-body-final} \cdot \text{r} \cdot \text{Throw} \cdot \text{s} \cdot f = f \]
\[ \text{eval-body-final} \cdot \text{r} \cdot (\text{Catch} \cdot x \cdot y) \cdot \text{s} \cdot f = \text{r} \cdot x \cdot \text{s} \cdot (\text{r} \cdot y \cdot \text{s} \cdot f) \]
\[ \text{eval-body-final-strictExpr} : \text{eval-body-final} \cdot \text{r} \cdot \text{\_} \cdot \text{s} \cdot f = \bot \]
\[ \text{eval-body-final-strictExpr} \cdot \text{r} \cdot \bot \cdot \text{s} \cdot f = \bot \]
\[ \text{eval-body-final-strictExpr} \cdot \text{r} \cdot \text{\_} \cdot \text{s} \cdot f = \bot \]
\[ \text{eval-body-final-strictExpr} \cdot \text{r} \cdot \bot \cdot \text{s} \cdot f = \bot \]
\[ \text{eval-body-final-strictExpr} \cdot \text{r} \cdot \text{\_} \cdot \text{s} \cdot f = \bot \]
\[ \text{eval-body-final-strictExpr} \cdot \text{r} \cdot \bot \cdot \text{s} \cdot f = \bot \]
\[ \text{eval-body-final-strictExpr} \cdot \text{r} \cdot \text{\_} \cdot \text{s} \cdot f = \bot \]
\[ \text{eval-body-final-strictExpr} \cdot \text{r} \cdot \bot \cdot \text{s} \cdot f = \bot \]
\[ \text{eval-body-final-strictExpr} \cdot \text{r} \cdot \text{\_} \cdot \text{s} \cdot f = \bot \]
\[ \text{eval-body-final-strictExpr} \cdot \text{r} \cdot \bot \cdot \text{s} \cdot f = \bot \]
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 llist =
\[ \text{lnil} \]
\[ \text{lcons (lazy 'a) (lazy 'a list)} \]

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

The following standard list functions underpin the monadic infrastructure:

\[ \text{fixrec lappend :: 'a list } \rightarrow \text{ 'a list } \rightarrow \text{ 'a list where} \]
\[ \text{lappend} \cdot \text{lnil} \cdot \text{ys} = \text{ys} \]
\[ \text{lappend} \cdot \text{lcons} \cdot \text{x} \cdot \text{xs} \cdot \text{ys} = \text{lcons} \cdot \text{x} \cdot \text{lappend} \cdot \text{xs} \cdot \text{ys} \]

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

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

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

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

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

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

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:

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

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

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

We interpret our language using these combinators in the obvious way. The only complication is that, even though our evaluator is primitive recursive, we must explicitly use the fixed point operator as the worker/wrapper technique requires us to talk about the body of the recursive definition.
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
\[
\begin{align*}
\text{evalK-body} & \equiv \Lambda r e. \text{case undiscr e of} \\
& \quad \text{const n} \Rightarrow \text{returnK} (\text{Discr n}) \\
& \quad \text{add e1 e2} \Rightarrow \text{addK} (r \cdot (\text{Discr e1})) \cdot (r \cdot (\text{Discr e2})) \\
& \quad \text{disj e1 e2} \Rightarrow \text{disjK} (r \cdot (\text{Discr e1})) \cdot (r \cdot (\text{Discr e2})) \\
& \quad \text{fail} \Rightarrow \text{failK}
\end{align*}
\]

abbreviation evalK :: 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 unwrap is the following function that converts a lazy list into an equivalent continuation representation.

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

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

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

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

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

The worker/wrapper condition follows directly from these definitions.

lemma KS-SK-id: 
\[
KS \cdot (SK \cdot xs) = xs
\]
by (induct xs) (simp-all add: KS-def failK-def)

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

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

lemma cfun-not-below: 
\[
f \cdot x \not\sqsubseteq g \cdot x \Rightarrow f \not\sqsubseteq g
\]
by (auto simp: cfun-below-iff)

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

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

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

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

\textbf{definition eval-ww :: expr discr \rightarrow nat discr llist where}
\texttt{eval-ww \equiv wrap \cdot 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 \(SK\) witnesses a \textit{monad morphism} (Wadler 1992, \S 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 SK-returnS-returnK:}
SK \cdot (\text{returnS} \cdot x) = \text{returnK} \cdot x
by (simp add: returnS-def returnK-def failK-def)

\textbf{lemma SK-lappend-distrib:}
SK \cdot (\text{lappend} \cdot x \cdot y) \cdot s \cdot f = SK \cdot x \cdot s \cdot (SK \cdot y \cdot s \cdot f)
by (induct xs) (simp-all add: failK-def)

\textbf{lemma SK-bindS-bindK:}
SK \cdot (\text{bindS} \cdot x \cdot g) = \text{bindK} \cdot (SK \cdot x) \cdot (SK \circ g)
by (induct x)
  (simp-all add: cfun-eq-iff
    bindS-def bindK-def failK-def
    SK-lappend-distrib)

\textbf{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)

\textbf{lemma SK-disjS-disjK:}
\begin{align*}
SK \cdot (\text{disjS} \cdot xs \cdot ys) &= \text{disjK} \cdot (SK \cdot xs) \cdot (SK \cdot ys) \\
\text{by (simp add: cfun-eq-iff disjS-def disjK-def SK-lappend-distrib)}
\end{align*}

**Lemma** \(SK\cdot\text{failS}=\text{failK}\)

**Unfolding** \(\text{failS-def by simp}\)

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

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

By (cases e)

\[(\text{simp-all add: evalS-body-def evalK-body-def unwrap-def} \text{SK-returnS-returnK SK-addS-distrib} \text{SK-disjS-disjK SK-failS-failK})\]

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

By simp

**Theorem** \(\text{evalS-evalK}:\)

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

**Using** worker-ccf-fusion-new[OF wrap-unwrap-id unwrap-strict]

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

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 nub function.

```haskell
fixrec nub :: Nat list → Nat list
where
  nub-nil = nil
| nub-(x :@ xs) = x :@ nub-\( \text{filter}(\Lambda\ y. \ x =B\ y) \)·xs)
```

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

```haskell
fixrec nub-body :: \( \text{Nat list} \rightarrow \text{Nat list} \) \rightarrow \text{Nat list} 
where
  nub-body·f-nil = nil
| nub-body·f·(x :@ xs) = x :@ f·(\text{filter}(\Lambda\ y. \ x =B\ y) \)·xs)
```

**Lemma nub-nub-body-eq**: \( \text{nub} = \text{fix}·\text{nub-body} \)
by (rule cfun-eqI, subst nub-def, subst nub-body, unfold, simp)

10.2 Optimised data type.

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

**Type-synonym** \( \text{NatSet} = \text{Nat list} \)

**Definition**

\( \text{SetEmpty} :: \text{NatSet} \) where
\( \text{SetEmpty} \equiv \text{nil} \)

**Definition**

\( \text{SetInsert} :: \text{Nat} \rightarrow \text{NatSet} \rightarrow \text{NatSet} \) where
\( \text{SetInsert} \equiv \text{icons} \)

**Definition**

\( \text{SetMem} :: \text{Nat} \rightarrow \text{NatSet} \rightarrow \text{tr} \) where
\( \text{SetMem} \equiv \text{inmember}(\text{bpred}(\text{op}=)) \)

**Lemma SetMem-strict[simp]**: \( \text{SetMem}·x·\bot = \bot \)
by (simp add: SetMem-def)

**Lemma SetMem-SetEmpty[simp]**: \( \text{SetMem}·x·\text{SetEmpty} = \text{FF} \)
by (simp add: SetMem-def SetEmpty-def)

**Lemma SetMem-SetInsert**: \( \text{SetMem}·v·(\text{SetInsert}·x·s) = (\text{SetMem}·v·s \text{ orelse } x =B\ v) \)
by (simp add: SetMem-def SetInsert-def)

AndyG’s new type.

**Domain** \( R = R \) (lazy \( \text{resultR} :: \text{Nat list} \)) (lazy \( \text{exceptR} :: \text{NatSet} \))

**Definition**

\( \text{nextR} :: R \rightarrow (\text{Nat} \times R) \) Maybe where
\( \text{nextR} = (\Lambda\ r. \ case\ \text{idropWhile}(\Lambda\ x. \text{SetMem}·x·(\text{exceptR}·r)·(\text{resultR}·r))\ of \)
\text{lnil} \Rightarrow \text{Nothing} \\
| x : :@ \text{xs} \Rightarrow \text{Just}(x, R\cdot\text{xs} \cdot (\text{exceptR}\cdot r))

\text{lemma nextR-strict1}[\text{simp}]: \text{nextR}\cdot \bot = \bot \text{ by (simp add: nextR-def)}
\text{lemma nextR-strict2}[\text{simp}]: \text{nextR}\cdot (R\cdot \bot \cdot S) = \bot \text{ by (simp add: nextR-def)}

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

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

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

\text{definition a2c} :: R \rightarrow \text{Nat llist} \text{ where}
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)

\text{lemma a2c-strict}[\text{simp}]: a2c \cdot \bot = \bot \text{ unfolding a2c-def by simp}
\text{lemma a2c-c2a-id}:
a2c \circ c2a = \text{ID} \text{ by (rule cfun-eqI, simp add: a2c-def c2a-def lfilter-const-true)}

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

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

\text{lemma unwrap-strict}[\text{simp}]: \text{unwrap} \cdot \bot = \bot \text{ unfolding unwrap-def by (rule cfun-eqI, simp)}
\text{lemma wrap-unwrap-id}:
wrap \circ \text{unwrap} = \text{ID} \text{ using cfun-fun-cong[OF a2c-c2a-id]}
\text{by} \quad ((\text{rule cfun-eqI})+, \text{ simp add: unwrap-def wrap-def})

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

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

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

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

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

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

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

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

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

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

lemma nub-body''-nub-body'''-eq: nub-body'' = nub-body'''
  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\cdot \bot = \bot

by fixrec-simp

fixrec llast-body :: ('a list \to 'a) \to 'a llist \to 'a
where
\llast-body.f.(x :@ yys) = (case yys of lnil \Rightarrow x | y :@ ys \Rightarrow f\cdot yys)

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

definition wrap :: ('a \to 'a llist) \to ('a llist \to 'a) where
wrap \equiv \Lambda f \cdot (x :@ xs) \cdot f\cdot x\cdot xs

definition unwrap :: ('a llist \to 'a) \to ('a \to 'a llist) \to 'a where
unwrap \equiv \Lambda f x xs \cdot f\cdot (x :@ xs)

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

lemma wrap-unwrap-ID: \ wrap \circ unwrap \circ \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 \to 'a llist \to 'a) \to 'a \to 'a llist \to 'a where
llast-worker \equiv \Lambda r x yys \cdot case yys of lnil \Rightarrow x | y :@ ys \Rightarrow r\cdot y\cdot ys

definition llast' :: 'a llist \to 'a where
llast' \equiv \Lambda \cdot \fix\cdot \llast-worker

lemma llast-worker-llast-body: \llast-worker = \unwrap \circ \llast-body \circ \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\cdot \llast-body by (simp only: llast-llast-body)
also have \ldots = \wrap\cdot \fix\cdot (\unwrap \circ \llast-body \circ \wrap)
  by (simp only: worker-wrapper-body[OF wrap-unwrap-ID])
also have \ldots = \wrap\cdot \fix\cdot (\llast-worker)
  by (simp only: llast-worker-llast-body)
also have \ldots = ?lhs unfolding llast'-def by simp
finally show ?thesis by simp

39
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


