

Mechanising the worker/wrapper transformation

Peter Gammie

February 23, 2021

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 <i>unwrap</i> 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 $O(n^2)$ nub into an $O(n \lg n)$ one	34
10.1 The nub function.	34
10.2 Optimised data type.	35
11 Optimise “last”.	38
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, $\Lambda_.$ for continuous function abstraction, $_ \cdot _$ for continuous function application, **domain** for recursive-datatype definition.

2 Fixed-point theorems for program transformation

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

2.1 The rolling rule

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

lemma *rolling-rule-ltr*: $fix \cdot (g \ oo \ f) \sqsubseteq g \cdot (fix \cdot (f \ oo \ g))$

proof –

have $g \cdot (fix \cdot (f \ oo \ g)) \sqsubseteq g \cdot (fix \cdot (f \ oo \ g))$

by (*rule below-refl*) — reflexivity

hence $g \cdot ((f \ oo \ g) \cdot (fix \cdot (f \ oo \ g))) \sqsubseteq g \cdot (fix \cdot (f \ oo \ g))$

using *fix-eq[where F=f oo g]* **by** *simp* — computation

hence $(g \ oo \ f) \cdot (g \cdot (fix \cdot (f \ oo \ g))) \sqsubseteq g \cdot (fix \cdot (f \ oo \ g))$

by *simp* — re-associate (*oo*)

thus $fix \cdot (g \ oo \ f) \sqsubseteq g \cdot (fix \cdot (f \ oo \ g))$

using *fix-least-below* by *blast* — induction
 qed

lemma *rolling-rule-rtl*: $g \cdot (\text{fix} \cdot (f \text{ oo } g)) \sqsubseteq \text{fix} \cdot (g \text{ oo } f)$

proof –

have $\text{fix} \cdot (f \text{ oo } g) \sqsubseteq f \cdot (\text{fix} \cdot (g \text{ oo } f))$ by (*rule rolling-rule-ltr*)

hence $g \cdot (\text{fix} \cdot (f \text{ oo } g)) \sqsubseteq g \cdot (f \cdot (\text{fix} \cdot (g \text{ oo } f)))$

by (*rule monofun-cfun-arg*) — g is monotonic

thus $g \cdot (\text{fix} \cdot (f \text{ oo } g)) \sqsubseteq \text{fix} \cdot (g \text{ oo } f)$

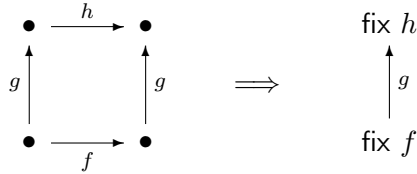
using *fix-eq*[**where** $F = g \text{ oo } f$] by *simp* — computation
 qed

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

by (*rule below-antisym*[*OF rolling-rule-ltr rolling-rule-rtl*])

2.2 Least-fixed-point fusion

Least-fixed-point fusion provides a kind of induction that has proven to be very useful in calculational settings. Intuitively it lifts the step-by-step correspondence between f and h witnessed by the strict function g to the fixed points of f and g :



Fokkinga and Meijer (1991), and also their later Meijer, Fokkinga, and Paterson (1991), made extensive use of this rule, as did Tullsen (2002) in his program transformation tool PATH. This diagram is strongly reminiscent of the simulations used to establish refinement relations between imperative programs and their specifications (de Roever and Engelhardt 1998).

The following proof is close to the third variant of Stoy (1977, p215). We relate the two fixpoints using the rule `parallel_fix_ind`:

$$\frac{\text{adm } (\lambda x. ?P (fst x) (snd x)) \quad ?P \perp \perp \quad \bigwedge x y. \frac{?P x y}{?P (?F \cdot x) (?G \cdot y)}}{?P (\text{fix} \cdot ?F) (\text{fix} \cdot ?G)}$$

in a very straightforward way:

lemma *lfp-fusion*:

assumes $g \cdot \perp = \perp$

assumes $g \text{ oo } f = h \text{ oo } g$

shows $g \cdot (\text{fix} \cdot f) = \text{fix} \cdot h$

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

case 2 show $g \cdot \perp = \perp$ **by fact**

For a recursive definition $comp = \text{fix } body$ for some $body :: A \rightarrow A$ and a pair of functions $wrap :: B \rightarrow A$ and $unwrap :: A \rightarrow B$ where $wrap \circ unwrap = id_A$, we have:

$$\begin{aligned} comp &= wrap \ work \\ work &:: B && \text{(the worker/wrapper} \\ work &= \text{fix } (unwrap \circ body \circ wrap) \end{aligned}$$

transformation)

Also:

$$(unwrap \circ wrap) \ work = work \quad \text{(worker/wrapper fusion)}$$

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

```

case ( $\exists x y$ )
from  $\langle g \cdot x = y \rangle \langle g \circ f = h \circ g \rangle$  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 $wrap \circ unwrap$ to be the identity for all values.

```

lemma worker-wrapper-id:
  fixes  $wrap :: 'b::pcpo \rightarrow 'a::pcpo$ 
  fixes  $unwrap :: 'a \rightarrow 'b$ 
  assumes wrap-unwrap:  $wrap \circ unwrap = ID$ 
  assumes comp-body:  $computation = \text{fix} \cdot body$ 
  shows  $computation = wrap \cdot (\text{fix} \cdot (unwrap \circ body \circ wrap))$ 
proof –
  from comp-body have  $computation = \text{fix} \cdot (ID \circ body)$ 
  by simp

```

```

also from wrap-unwrap have ... = fix·(wrap oo unwrap oo body)
  by (simp add: assoc-oo)
also have ... = wrap·(fix·(unwrap oo body oo wrap))
  using rolling-rule[where f=unwrap oo body and g=wrap]
  by (simp add: assoc-oo)
finally show ?thesis .
qed

```

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

```

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

```

This is particularly useful when the computation being transformed is strict in its argument.

Finally we can allow the identity to take the full recursive context into account. This rule was described by Gill and Hutton but not used.

```

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

```

Gill and Hutton’s *worker-wrapper-fusion* rule is intended to allow the transformation of $(\text{unwrap oo wrap}) \cdot R$ 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 wrap :: 'b::pcpo → 'a::pcpo
  fixes unwrap :: 'a → 'b
  assumes wrap-unwrap: wrap oo unwrap = ID
  assumes work: work = fix.(unwrap oo body oo wrap)
  shows (unwrap oo wrap)·work = work
proof –
  have (unwrap oo wrap)·work = (unwrap oo wrap)·(fix.(unwrap oo body oo wrap))
    using work by simp
  also have ... = (unwrap oo wrap)·(fix.(unwrap oo body oo wrap oo unwrap oo
wrap))
    using wrap-unwrap by (simp add: assoc-oo)
  also have ... = fix.(unwrap oo wrap oo unwrap oo body oo wrap)
    using rolling-rule[where f=unwrap oo body oo wrap and g=unwrap oo wrap]
    by (simp add: assoc-oo)
  also have ... = fix.(unwrap oo body oo wrap)
    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 $\text{body} \Longrightarrow f$, 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:

$$\frac{f \cdot ?x = ?x}{\text{fix} \cdot f \sqsubseteq ?x} \text{fix_least}$$

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 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 body' such that the body of the worker, i.e. the expression $\text{unwrap } \text{oo } \text{body } \text{oo } \text{wrap}$, can be rewritten as $\text{body}' \text{ oo } \text{unwrap } \text{oo } \text{wrap}$. Intuitively this is the semantic form of workers where all self-calls are fusible. Our goal is to justify redefining work to $\text{fix} \cdot \text{body}'$, i.e. to establish:

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

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

lemma *fusion-partially-correct*:

assumes *wrap-unwrap*: $\text{wrap } \text{oo } \text{unwrap} = \text{ID}$

assumes *work*: $\text{work} = \text{fix} \cdot (\text{unwrap } \text{oo } \text{body } \text{oo } \text{wrap})$

assumes *body'*: $\text{unwrap } \text{oo } \text{body } \text{oo } \text{wrap} = \text{body}' \text{ oo } \text{unwrap } \text{oo } \text{wrap}$

shows $\text{fix} \cdot \text{body}' \sqsubseteq \text{work}$

proof(*rule fix-least*)

have $\text{work} = (\text{unwrap } \text{oo } \text{body } \text{oo } \text{wrap}) \cdot \text{work}$

using *work* **by** (*simp add: fix-eq[symmetric]*)

also have $\dots = (\text{body}' \text{ oo } \text{unwrap } \text{oo } \text{wrap}) \cdot \text{work}$

using *body'* **by** *simp*

also have $\dots = (\text{body}' \text{ oo } \text{unwrap } \text{oo } \text{wrap}) \cdot ((\text{unwrap } \text{oo } \text{body } \text{oo } \text{wrap}) \cdot \text{work})$

using *work* **by** (*simp add: fix-eq[symmetric]*)

also have $\dots = (\text{body}' \text{ oo } \text{unwrap } \text{oo } \text{wrap } \text{oo } \text{unwrap } \text{oo } \text{body } \text{oo } \text{wrap}) \cdot \text{work}$

by *simp*

also have $\dots = (\text{body}' \text{ oo } \text{unwrap } \text{oo } \text{body } \text{oo } \text{wrap}) \cdot \text{work}$

using *wrap-unwrap* **by** (*simp add: assoc-oo*)

also have $\dots = \text{body}' \cdot \text{work}$

using *work* **by** (*simp add: fix-eq[symmetric]*)

finally show $\text{body}' \cdot \text{work} = \text{work}$ **by** *simp*

qed

The next section shows the converse does not obtain.

3.2 A non-strict unwrap may go awry

If 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 *wrap* and *unwrap* that map between these types are routine. Note that *wrap* is (necessarily) strict due to the property $\forall x. ?f.(?g.x) = x \implies ?f.\perp = \perp$.

fixrec *wrap* :: $B \rightarrow A$
where *wrap*·($B \cdot a$) = a

fixrec *unwrap* :: $A \rightarrow B$
where *unwrap* = B

Discharging the worker/wrapper hypothesis is similarly routine.

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.

fixrec *body* :: $A \rightarrow A$
where *body*· r = A

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

fixrec *body'* :: $B \rightarrow B$
where *body'*·($B \cdot a$) = $B \cdot A$

As explained above, we set up the fusion opportunity:

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:

lemma *fix-body'* \sqsubseteq *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:

lemma \neg *fix*·(*unwrap oo body oo wrap*) \sqsubseteq *fix*·*body'*

proof –

have l : *fix*·(*unwrap oo body oo wrap*) = $B \cdot A$

by (*subst fix-eq simp*)

have r : *fix*·*body'* = \perp

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 $wrap \circ unwrap = ID$, then making $unwrap$ strict preserves this equation:

```

lemma
  assumes  $wrap \circ unwrap = ID$ 
  shows  $wrap \circ strictify \cdot unwrap = ID$ 
proof(rule cfun-eqI)
  fix  $x$ 
  from assms
  show  $(wrap \circ strictify \cdot unwrap) \cdot x = ID \cdot x$ 
    by (cases  $x = \perp$ ) (simp-all add: cfun-eq-iff retraction-strict)
qed

```

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

4 A totally-correct fusion rule

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

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

```

lemma worker-wrapper-fusion-new:
  fixes  $wrap :: 'b::pcpo \rightarrow 'a::pcpo$ 
  fixes  $unwrap :: 'a \rightarrow 'b$ 
  fixes  $body' :: 'b \rightarrow 'b$ 
  assumes  $wrap \cdot unwrap: wrap \circ unwrap = (ID :: 'a \rightarrow 'a)$ 
  assumes  $unwrap \cdot strict: unwrap \cdot \perp = \perp$ 
  assumes  $body \cdot body': unwrap \circ body \circ wrap = body' \circ (unwrap \circ wrap)$ 
  shows  $fix \cdot body = wrap \cdot (fix \cdot body')$ 
proof –
  from  $body \cdot body'$ 
  have  $unwrap \circ body \circ (wrap \circ unwrap) = (body' \circ unwrap \circ (wrap \circ unwrap))$ 
    by (simp add: assoc-oo)
  with  $wrap \cdot unwrap$  have  $unwrap \circ body = body' \circ unwrap$ 
    by simp
  with  $unwrap \cdot strict$  have  $unwrap \cdot (fix \cdot body) = fix \cdot body'$ 
    by (rule lfp-fusion)
  hence  $(wrap \circ unwrap) \cdot (fix \cdot body) = wrap \cdot (fix \cdot body')$ 
    by simp
  with  $wrap \cdot unwrap$  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 \rightarrow 'a::pcpo

fixes *unwrap* :: 'a \rightarrow 'b

assumes *wrap-unwrap*: *wrap* oo *unwrap* = (*ID* :: 'a \rightarrow 'a)

assumes *unwrap-strict*: *unwrap*. \perp = \perp

assumes *body-body'*: $\bigwedge r. (unwrap \text{ oo } wrap) \cdot r = r$
 $\implies (unwrap \text{ oo } body \text{ oo } wrap) \cdot r = body' \cdot r$

shows *fix*.*body* = *wrap*.(*fix*.*body'*)

proof –

let *?P* = $\lambda(x, y). x = y \wedge unwrap \cdot (wrap \cdot x) = x$

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* (\perp , \perp)

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 *llist* \rightarrow 'a *llist*

definition

For a recursive definition $comp = body$ of type A and a pair of functions $wrap :: B \rightarrow A$ and $unwrap :: A \rightarrow B$ where $wrap \circ unwrap = id_A$ and $unwrap \perp = \perp$, define:

$$\begin{aligned} comp &= wrap \ work && \text{(the worker/wrapper} \\ work &= unwrap (body[wrap work/comp]) \end{aligned}$$

transformation)

In the scope of $work$, the following rewrite is admissable:

$$unwrap (wrap work) \Longrightarrow work \quad \text{(worker/wrapper fusion)}$$

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

$list2H :: 'a \text{ llist} \rightarrow 'a \ H \ \mathbf{where}$
 $list2H \equiv lappend$

lemma *acc-c2a-strict[simp]*: $list2H.\perp = \perp$
by (*rule cfun-eqI, simp add: list2H-def*)

definition

$H2list :: 'a \ H \rightarrow 'a \ \text{llist} \ \mathbf{where}$
 $H2list \equiv \Lambda f . f.\text{lnil}$

The paper only claims the homomorphism holds for finite lists, but in fact it holds for all lazy lists in HOLCF. They are trying to dodge an explicit appeal to the equation $\perp = (\Lambda x . \perp)$, which does not hold in Haskell.

lemma *H-llist-hom-append*: $list2H.(xs :++ ys) = list2H.xs \text{ oo } list2H.ys$ (**is** $?lhs = ?rhs$)

proof(*rule cfun-eqI*)

fix zs

have $?lhs.zs = (xs :++ ys) :++ zs$ **by** (*simp add: list2H-def*)

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

also have $\dots = list2H.xs.(ys :++ zs)$ **by** (*simp add: list2H-def*)

also have $\dots = list2H.xs.(list2H.ys.zs)$ **by** (*simp add: list2H-def*)

also have $\dots = (list2H.xs \text{ oo } list2H.ys).zs$ **by** *simp*

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

qed

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

lemma *H2list-list2H-inv*: $H2list \text{ oo } list2H = ID$

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

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

fixrec $lrev :: 'a \ \text{llist} \rightarrow 'a \ \text{llist}$

where

$lrev.nil = nil$
| $lrev.(x :@ xs) = lrev.xs :++ (x :@ nil)$

Note “body” is the generator of $lrev-def$.

lemma $lrev-strict[simp]$: $lrev.\perp = \perp$
by $fixrec-simp$

fixrec $lrev-body :: ('a llist \rightarrow 'a llist) \rightarrow 'a llist \rightarrow 'a llist$

where

$lrev-body.r.nil = nil$
| $lrev-body.r.(x :@ xs) = r.xs :++ (x :@ nil)$

lemma $lrev-body-strict[simp]$: $lrev-body.r.\perp = \perp$
by $fixrec-simp$

This is trivial but syntactically a bit touchy. Would be nicer to define $lrev-body$ as the generator of the fixpoint definition of $lrev$ directly.

lemma $lrev-lrev-body-eq$: $lrev = fix.lrev-body$
by ($rule\ cfun-eqI$, $subst\ lrev-def$, $subst\ lrev-body.unfold$, $simp$)

Wrap / unwrap functions.

definition

$unwrapH :: ('a llist \rightarrow 'a llist) \rightarrow 'a llist \rightarrow 'a H$ **where**
 $unwrapH \equiv \Lambda f\ xs . list2H.(f.xs)$

lemma $unwrapH-strict[simp]$: $unwrapH.\perp = \perp$
unfolding $unwrapH-def$ **by** ($rule\ cfun-eqI$, $simp$)

definition

$wrapH :: ('a llist \rightarrow 'a H) \rightarrow 'a llist \rightarrow 'a llist$ **where**
 $wrapH \equiv \Lambda f\ xs . H2list.(f.xs)$

lemma $wrapH-unwrapH-id$: $wrapH\ oo\ unwrapH = ID$ (**is** $?lhs = ?rhs$)

proof($rule\ cfun-eqI$)**+**

fix $f\ xs$

have $?lhs.f.xs = H2list.(list2H.(f.xs))$ **by** ($simp\ add$: $wrapH-def\ unwrapH-def$)

also have $\dots = (H2list\ oo\ list2H).(f.xs)$ **by** $simp$

also have $\dots = ID.(f.xs)$ **by** ($simp\ only$: $H2list-list2H-inv$)

also have $\dots = ?rhs.f.xs$ **by** $simp$

finally show $?lhs.f.xs = ?rhs.f.xs$.

qed

5.2 Gill/Hutton-style worker/wrapper.

definition

$lrev-work :: 'a llist \rightarrow 'a H$ **where**
 $lrev-work \equiv fix.(unwrapH\ oo\ lrev-body\ oo\ wrapH)$

definition

$lrev-wrap :: 'a\ llist \rightarrow 'a\ llist$ **where**
 $lrev-wrap \equiv wrapH \cdot lrev-work$

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 $lrev-body1 :: ('a\ llist \rightarrow 'a\ H) \rightarrow 'a\ llist \rightarrow 'a\ H$

where

$lrev-body1 \cdot r \cdot lnil = list2H \cdot lnil$
 $| lrev-body1 \cdot r \cdot (x :@ xs) = list2H \cdot (wrapH \cdot r \cdot xs :++ (x :@ lnil))$

definition

$lrev-work1 :: 'a\ llist \rightarrow 'a\ H$ **where**
 $lrev-work1 \equiv fix \cdot lrev-body1$

lemma $lrev-body-lrev-body1-eq$: $lrev-body1 = unwrapH \circ lrev-body \circ wrapH$

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 $lrev-body2 :: ('a\ llist \rightarrow 'a\ H) \rightarrow 'a\ llist \rightarrow 'a\ H$

where

$lrev-body2 \cdot r \cdot lnil = ID$
 $| lrev-body2 \cdot r \cdot (x :@ xs) = list2H \cdot (wrapH \cdot r \cdot xs) \circ list2H \cdot (x :@ lnil)$

lemma $lrev-body2-strict[simp]$: $lrev-body2 \cdot r \cdot \perp = \perp$

by $fixrec-simp$

definition

$lrev-work2 :: 'a\ llist \rightarrow 'a\ H$ **where**
 $lrev-work2 \equiv fix \cdot lrev-body2$

lemma $lrev-work2-strict[simp]$: $lrev-work2 \cdot \perp = \perp$

unfolding $lrev-work2-def$
by ($subst\ fix-eq$) $simp$

lemma *rev-body2-rev-body1-eq*: *rev-body2* = *rev-body1*
by ((*rule cfun-eqI*)
, (*subst rev-body1.unfold, subst rev-body2.unfold*)
, (*simp add: H-llist-hom-append[symmetric] H-llist-hom-id*))

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

Simplify.

fixrec *rev-body3* :: ('a llist → 'a H) → 'a llist → 'a H
where
rev-body3.r.nil = *ID*
| *rev-body3.r.(x :@ xs)* = *r.xs oo list2H.(x :@ nil)*

lemma *rev-body3-strict[simp]*: *rev-body3.r.⊥* = ⊥
by *fixrec-simp*

definition

rev-work3 :: 'a llist → 'a H **where**
rev-work3 ≡ *fix.rev-body3*

lemma *rev-wwfusion*: *list2H.(wrapH.rev-work2).xs* = *rev-work2.xs*

proof –

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

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

lemma *rev-work3* ⊆ *rev-work2*

unfolding *rev-work3-def*

proof(*rule fix-least*)

{
fix *xs* **have** *rev-body3.rev-work2.xs* = *rev-work2.xs*
proof(*cases xs*)
case bottom **thus** *?thesis* **by** *simp*

```

next
  case lnil thus ?thesis
    unfolding lev-work2-def
    by (subst fix-eq[where F=lev-body2], simp)
next
  case (lcons y ys)
  hence lev-body3.lev-work2.xs = lev-work2.yz oo list2H.(y :@ lnil) by simp
  also have  $\dots = list2H.((wrapH.lev-work2).yz) oo list2H.(y :@ lnil)$ 
    using lev-wwfusion[where xs=yz] by simp
  also from lcons have  $\dots = lev-body2.lev-work2.xs$  by simp
  also have  $\dots = lev-work2.xs$ 
    unfolding lev-work2-def by (simp only: fix-eq[symmetric])
  finally show ?thesis by simp
qed
}
thus lev-body3.lev-work2 = lev-work2 by (rule cfun-eqI)
qed

```

We can't show the reverse inclusion in the same way as the fusion law doesn't hold for the optimised definition. (Intuitively we haven't established that it is equal to the original *lev* 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.

lemma *lev-work3-lev-work2-eq: lev-work3 = lev-work2 (is ?lhs = ?rhs)*

proof(*rule cfun-eqI*)

fix *x*

show *?lhs.x = ?rhs.x*

proof(*induct x*)

show *lev-work3.⊥ = lev-work2.⊥*

apply (*unfold lev-work3-def lev-work2-def*)

apply (*subst fix-eq[where F=lev-body2]*)

apply (*subst fix-eq[where F=lev-body3]*)

by (*simp add: lev-body3.unfold lev-body2.unfold*)

next

show *lev-work3.lnil = lev-work2.lnil*

apply (*unfold lev-work3-def lev-work2-def*)

apply (*subst fix-eq[where F=lev-body2]*)

apply (*subst fix-eq[where F=lev-body3]*)

by (*simp add: lev-body3.unfold lev-body2.unfold*)

next

fix *a l* **assume** *lev-work3.l = lev-work2.l*

thus *lev-work3.(a :@ l) = lev-work2.(a :@ l)*

apply (*unfold lev-work3-def lev-work2-def*)

apply (*subst fix-eq[where F=lev-body2]*)

apply (*subst fix-eq[where F=lev-body3]*)

```

  apply (fold lrev-work3-def lrev-work2-def)
  apply (simp add: lrev-body3.unfold lrev-body2.unfold lrev-wwfusion)
done
qed simp-all
qed

```

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

```

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

```

```

lemma lrev-work3-lrev-work2-eq': lrev = wrapH · lrev-work3
proof –
  from lrev-lrev-body-eq
  have lrev = fix · lrev-body .
  also from wrapH-unwrapH-id unwrapH-strict
  have ... = 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 lev-final-lev-eq: lev = lev-final (is ?lhs = ?rhs)
proof –
  have ?lhs = lev-wrap by (rule lev-lev-ww-eq)
  also have ... = wrapH·lev-work by (simp only: lev-wrap-def)
  also have ... = wrapH·lev-work1 by (simp only: lev-work1-lev-work-eq)
  also have ... = wrapH·lev-work2 by (simp only: lev-work2-lev-work1-eq)
  also have ... = wrapH·lev-work3 by (simp only: lev-work3-lev-work2-eq)
  also have ... = wrapH·lev-work-final by (simp only: lev-work3-def lev-work-final-def
lev-body-final-lev-body3-eq)
  also have ... = lev-final by (simp add: lev-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, $UNat_{\perp}$ the lifted (flat and pointed) variant, and Nat the standard boxed domain, isomorphic to $UNat_{\perp}$. 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· $\perp$  =  $\perp$ 
```

```
by fixrec-simp
```

```
definition
```

```
  fac-body :: (Nat → Nat) → Nat → Nat where
```

```
  fac-body ≡  $\Lambda$  r n. If n =B 0 then 1 else n * r·(n – 1)
```

```
lemma fac-body-strict[simp]: fac-body·r· $\perp$  =  $\perp$ 
```

```
  unfolding fac-body-def by simp
```

lemma *fac-fac-body-eq*: $fac = fix \cdot fac\text{-}body$
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

$unwrapB :: (Nat \rightarrow Nat) \rightarrow UNat \rightarrow UNat_{\perp}$ **where**
 $unwrapB \equiv \Lambda f . unbox \circ f \circ box$

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

definition

$wrapB :: (UNat \rightarrow UNat_{\perp}) \rightarrow Nat \rightarrow Nat$ **where**
 $wrapB \equiv \Lambda f x . unbox \cdot x >>= f >>= box$

lemma *wrapB-unwrapB-body*:

assumes *strictF*: $f \cdot \perp = \perp$
shows $(wrapB \circ unwrapB) \cdot f = f$ (**is** *?lhs = ?rhs*)

proof(*rule cfun-eqI*)

fix $x :: Nat$

have $?lhs \cdot x = unbox \cdot x >>= (\Lambda x' . unwrapB \cdot f \cdot x' >>= box)$

unfolding *wrapB-def* **by** *simp*

also have $\dots = unbox \cdot x >>= (\Lambda x' . unbox \cdot (f \cdot (box \cdot x')) >>= box)$

unfolding *unwrapB-def* **by** *simp*

also from *strictF* **have** $\dots = f \cdot x$ **by** (*cases x*, *simp-all*)

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

qed

Apply worker/wrapper.

definition

$fac\text{-}work :: UNat \rightarrow UNat_{\perp}$ **where**
 $fac\text{-}work \equiv fix \cdot (unwrapB \circ fac\text{-}body \circ wrapB)$

definition

$fac\text{-}wrap :: Nat \rightarrow Nat$ **where**
 $fac\text{-}wrap \equiv wrapB \cdot fac\text{-}work$

lemma *fac-fac-wv-eq*: $fac = fac\text{-}wrap$ (**is** *?lhs = ?rhs*)

proof –

have $wrapB \circ unwrapB \circ fac\text{-}body = fac\text{-}body$

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' ≡ $\Lambda r n.$
 $unbox.(If\ box.n =_B\ 0$
 $\quad then\ 1$
 $\quad else\ unbox.(box.n - 1) >>= r >>= (\Lambda b. box.n * box.b))$

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=* $\Lambda y. box.x * y$ **and** *g=* $unbox.(box.x - 1)$]
 $bbind-case-distr-strict$ [**where** *f=* $\Lambda y. box.x * y$ **and** *h=* box]
unfolding *fac-body'-def fac-body-def unwrapB-def wrapB-def* **by** *simp*
qed

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

definition

fac-body-final :: (*UNat* → *UNat*_⊥) → *UNat* → *UNat*_⊥ **where**
fac-body-final ≡ $\Lambda r n.$
 $if\ n = 0\ then\ up.1\ else\ r.(n -_{\#}\ 1) >>= (\Lambda b. up.(n *_{\#}\ b))$

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

proof(*rule cfun-eqI*)+

fix *r x*
show *?lhs.r.x = ?rhs.r.x*
using *bbind-case-distr-strict*[**where** *f=* $unbox$ **and** *g=* $r.(x -_{\#}\ 1)$ **and** *h=* $(\Lambda b. box.(x *_{\#}\ b))$]
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 ≡ $\Lambda n. unbox.n >>= fac-work-final >>= box$

lemma *fac-fac-final*: $fac = fac\text{-}final$ (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 ... = *wrapB*·(*fix*·*fac-body'*) **by** (*simp only: fac-body'-fac-body*)
also have ... = *wrapB*·*fac-work-final* **by** (*simp only: fac-body-final-fac-body'*)
fac-work-final-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 ($\>=>$), in the homomorphism.

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

type-synonym $UNatAcc = UNat \rightarrow UNat_{\perp}$

definition

$n2a :: UNat \rightarrow UNatAcc$ **where**
 $n2a \equiv \Lambda m n. up \cdot (m \text{ *}_{\#} n)$

definition

$a2n :: UNatAcc \rightarrow UNat_{\perp}$ **where**
 $a2n \equiv \Lambda a. a \cdot 1$

lemma *a2n-strict[simp]*: $a2n \cdot \perp = \perp$

unfolding *a2n-def* **by** *simp*

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

unfolding *a2n-def n2a-def* **by** (*simp add: uMult-arithmetic*)

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

unfolding *n2a-def bKleisli-def* **by** (*simp add: uMult-arithmetic*)

definition

$unwrapA :: (UNat \rightarrow UNat_{\perp}) \rightarrow UNat \rightarrow UNatAcc$ **where**
 $unwrapA \equiv \Lambda f n. f \cdot n \>>= n2a$

lemma *unwrapA-strict[simp]*: $unwrapA \cdot \perp = \perp$

unfolding *unwrapA-def* **by** (rule *cfun-eqI*) *simp*

definition

$wrapA :: (UNat \rightarrow UNatAcc) \rightarrow UNat \rightarrow UNat_{\perp}$ **where**
 $wrapA \equiv \Lambda f. a2n \text{ oo } f$

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

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 - $body1 :: (UNat \rightarrow UNatAcc) \rightarrow UNat \rightarrow UNatAcc$ **where**

fac - acc - $body1 \equiv \Lambda r n.$

$if\ n = 0\ then\ n2a.1\ else\ wrapA.r.(n -\# 1) >>= (\Lambda\ res.\ n2a.(n *_{\#}\ res))$

lemma fac - acc - $body1$ - fac - $body$ - $final$ - eq : fac - acc - $body1 = unwrapA \text{ oo } fac$ - $body$ - $final$
 $oo\ wrapA$

unfolding fac - acc - $body1$ - def fac - $body$ - $final$ - def $wrapA$ - def $unwrapA$ - def

by ($rule$ $cfun$ - eqI) $+$ $simp$

Use the homomorphism.

definition

fac - acc - $body2 :: (UNat \rightarrow UNatAcc) \rightarrow UNat \rightarrow UNatAcc$ **where**

fac - acc - $body2 \equiv \Lambda r n.$

$if\ n = 0\ then\ n2a.1\ else\ wrapA.r.(n -\# 1) >>= (\Lambda\ res.\ n2a.n >=> n2a.res)$

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 - $body3 :: (UNat \rightarrow UNatAcc) \rightarrow UNat \rightarrow UNatAcc$ **where**

fac - acc - $body3 \equiv \Lambda r n.$

$if\ n = 0\ then\ n2a.1\ else\ n2a.n >=> r.(n -\# 1)$

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

proof($rule$ $cfun$ - eqI) $+$

fix $r\ n\ acc$

show $((fac$ - acc - $body3 \text{ oo } (unwrapA \text{ oo } wrapA)).r.n.acc) = fac$ - acc - $body2.r.n.acc$

unfolding fac - acc - $body2$ - def fac - acc - $body3$ - def $unwrapA$ - def

using $bbind$ - $case$ - $distr$ - $strict$ [**where** $f = \Lambda y. n2a.n >=> y$ **and** $h = n2a$, sym - $metric$]

by $simp$

qed

lemma *fac-work-final-body3-eq*: $fac\text{-}work\text{-}final = wrapA.(fix\text{-}fac\text{-}acc\text{-}body3)$
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

$fac\text{-}acc\text{-}body\text{-}final :: (UNat \rightarrow UNatAcc) \rightarrow UNat \rightarrow UNatAcc$ **where**
 $fac\text{-}acc\text{-}body\text{-}final \equiv \Lambda r\ n\ acc.$
if $n = 0$ *then* $up\text{-}acc$ *else* $r.(n -\# 1).(n *\# acc)$

definition

$fac\text{-}acc\text{-}work\text{-}final :: UNat \rightarrow UNat_{\perp}$ **where**
 $fac\text{-}acc\text{-}work\text{-}final \equiv \Lambda x. fix\text{-}fac\text{-}acc\text{-}body\text{-}final.x.1$

lemma *fac-acc-work-final-fac-acc-work3-eq*: $fac\text{-}acc\text{-}body\text{-}final = fac\text{-}acc\text{-}body3$ (**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*: $fac\text{-}acc\text{-}work\text{-}final = fac\text{-}work\text{-}final$ (**is** *?lhs=?rhs*)

proof –

have *?rhs* = $wrapA.(fix\text{-}fac\text{-}acc\text{-}body3)$ **by** (*rule fac-work-final-body3-eq*)
also have $\dots = wrapA.(fix\text{-}fac\text{-}acc\text{-}body\text{-}final)$
using *fac-acc-work-final-fac-acc-work3-eq* **by** *simp*
also have $\dots = ?lhs$
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* :: $('a \rightarrow 'b) \rightarrow 'a\ Stream \rightarrow 'b\ Stream$
where

$smap.f.(x \&\& xs) = f.x \&\& smap.f.xs$

lemma *smap-smap*: $smap.f.(smap.g.xs) = smap.(f \circ\circ g).xs$

fixrec *i-th* :: $'a\ Stream \rightarrow Nat \rightarrow 'a$

where

$$i\text{-th}\cdot(x \ \&\& \ xs) = \text{Nat-case}\cdot x\cdot(i\text{-th}\cdot xs)$$

abbreviation

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

The infinite stream of natural numbers.

fixrec $nats :: \text{Nat} \ \text{Stream}$

where

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

7.2 The wrapper/unwrapper functions.

definition

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

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

fixrec $unwrapS :: (\text{Nat} \rightarrow 'a) \rightarrow 'a \ \text{Stream}$

where

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

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

proof(*rule cfun-eqI*)

fix f **show** ?lhs $\cdot f = ?rhs\cdot f$

proof(*coinduct rule: Stream.coinduct*)

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

show *Stream-bisim* ? R

proof

fix $s \ s'$ **assume** ? $R \ s \ s'$

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

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

by *blast*

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

by (*rule exI*[**where** $x=f \ \text{oo} \ (\Lambda \ x. \ 1 + x)$]

, *subst* $unwrapS\text{-unfold}$, *subst* $nats\text{-unfold}$, *simp* *add*: $\text{smap}\text{-smap}$)

with $fs \ fs'$

show $(s = \perp \wedge s' = \perp)$

$\vee (\exists h \ t \ t'.$

$(\exists f. t = f\cdot 0 \ \&\& \ unwrapS\cdot(f \ \text{oo} \ (\Lambda \ x. \ 1 + x))$

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

$\wedge s = h \ \&\& \ t \wedge s' = h \ \&\& \ t')$ **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
qed
qed

```

definition

```

wrapS :: 'a Stream → Nat → 'a where
wrapS ≡ Λ s i . s !! i

```

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

In practice all functions worth memoising are strict in the memoised 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*)

```

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 ... = unwrapS·(f oo (Λ x. 1 + x)) !! i by simp

```

```

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

```

```

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

```

```

finally show ?case .

```

qed

lemma *wrapS-unwrapS-id*: $f·⊥ = ⊥ \implies (wrapS \text{ oo } unwrapS)·f = f$

```

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

```

7.3 Fibonacci example.

definition

```

fib-body :: (Nat → Nat) → Nat → Nat where

```

```

fib-body ≡ Λ r. Nat-case·1·(Nat-case·1·(Λ n. r·n + r·(n + 1)))

```

definition

```

fib :: Nat → Nat where

```

```

fib ≡ fix·fib-body

```


Apply worker/wrapper.

definition

fib-work :: *Nat Stream* **where**
fib-work \equiv *fix*·(*unwrapS* oo *fib-body* oo *wrapS*)

definition

fib-wrap :: *Nat* \rightarrow *Nat* **where**
fib-wrap \equiv *wrapS*·*fib-work*

lemma *wrapS-unwrapS-fib-body*: *wrapS* oo *unwrapS* oo *fib-body* = *fib-body*
proof(*rule cfun-eqI*)

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

lemma *fib-ww-eq*: *fib* = *fib-wrap*

using *worker-wrapper-body*[*OF wrapS-unwrapS-fib-body*]
by (*simp add: fib-def fib-wrap-def fib-work-def*)

Optimise.

fixrec

fib-work-final :: *Nat Stream*

and

fib-f-final :: *Nat* \rightarrow *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* \rightarrow *Nat* **where**
fib-final \equiv Λ *n*. *fib-work-final* !! *n*

This proof is only fiddly due to the way mutual recursion is encoded: we need to use Bekić's Theorem (Bekić 1984)¹ to massage the definitions into their final form.

lemma *fib-work-final-fib-work-eq*: *fib-work-final* = *fib-work* (**is** ?*lhs* = ?*rhs*)

proof –

let ?*wb* = Λ *r*. *Nat-case*·1·(*Nat-case*·1·(Λ *n'*. *r* !! *n'* + *r* !! (*n'* + 1)))

let ?*mr* = Λ (*fwf* :: *Nat Stream*, *fff*). (*smap*·*fff*·*nats*, ?*wb*·*fwf*)

have ?*lhs* = *fst* (*fix*·?*mr*)

by (*simp add: fib-work-final-def split-def csplit-def*)

¹The interested reader can find some historical commentary in Harel (1980); Sangiorgi (2009).

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

lemma *fib-final-fib-eq*: $\text{fib-final} = \text{fib}$ (**is** *?lhs = ?rhs*)

proof –

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

8 Tagless interpreter via double-barreled continuations

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

definition

val2cont $:: 'a \rightarrow 'a \text{Cont}$ **where**
val2cont $\equiv (\Lambda a c. c \cdot a)$

definition

cont2val $:: 'a \text{Cont} \rightarrow 'a$ **where**
cont2val $\equiv (\Lambda f. f \cdot \text{ID})$

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

domain *Expr* =

$\text{Val } (\text{lazy } \text{val} :: \text{Nat})$
 $| \text{Add } (\text{lazy } \text{addl} :: \text{Expr}) (\text{lazy } \text{addr} :: \text{Expr})$
 $| \text{Throw}$
 $| \text{Catch } (\text{lazy } \text{cbody} :: \text{Expr}) (\text{lazy } \text{handler} :: \text{Expr})$

fixrec *eval* $:: \text{Expr} \rightarrow \text{Nat Maybe}$

where

$\text{eval} \cdot (\text{Val} \cdot n) = \text{Just} \cdot n$
 $| \text{eval} \cdot (\text{Add} \cdot x \cdot y) = \text{mliftM2 } (\Lambda a b. a + b) \cdot (\text{eval} \cdot x) \cdot (\text{eval} \cdot y)$
 $| \text{eval} \cdot \text{Throw} = \text{mfail}$
 $| \text{eval} \cdot (\text{Catch} \cdot x \cdot y) = \text{mcatch} \cdot (\text{eval} \cdot x) \cdot (\text{eval} \cdot y)$

fixrec *eval-body* :: (*Expr* → *Nat Maybe*) → *Expr* → *Nat Maybe*

where

eval-body·*r*·(*Val*·*n*) = *Just*·*n*
| *eval-body*·*r*·(*Add*·*x*·*y*) = *mLiftM2* (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* → *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* → *Nat Maybe*)
→ *Expr* → (*Nat* → *Nat Maybe*) → *Nat Maybe* → *Nat Maybe*

where

eval-body'·*r*·(*Val*·*n*)·*s*·*f* = *s*·*n*
| *eval-body'*·*r*·(*Add*·*x*·*y*)·*s*·*f* = (*case wrapC*·*r*·*x* of

$$\begin{aligned}
& \text{Nothing} \Rightarrow f \\
& | \text{Just}\cdot n \Rightarrow (\text{case wrapC}\cdot r\cdot y \text{ of} \\
& \quad \text{Nothing} \Rightarrow f \\
& \quad | \text{Just}\cdot m \Rightarrow s\cdot(n + m))) \\
| \text{eval-body}'\cdot r\cdot \text{Throw}\cdot s\cdot f = f \\
| \text{eval-body}'\cdot r\cdot (\text{Catch}\cdot x\cdot y)\cdot s\cdot f = (\text{case wrapC}\cdot r\cdot x \text{ of} \\
& \quad \text{Nothing} \Rightarrow (\text{case wrapC}\cdot r\cdot y \text{ of} \\
& \quad \quad \text{Nothing} \Rightarrow f \\
& \quad \quad | \text{Just}\cdot n \Rightarrow s\cdot n) \\
& | \text{Just}\cdot n \Rightarrow s\cdot n)
\end{aligned}$$

lemma *eval-body'-strictExpr[simp]*: $\text{eval-body}'\cdot r\cdot \perp\cdot s\cdot f = \perp$
by (*subst eval-body'.unfold, simp*)

definition

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

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

lemma *eval-body'-eval-body-eq*: $\text{eval-body}' = \text{unwrapC} \circ \text{eval-body} \circ \text{wrapC}$
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* :: $(\text{Expr} \rightarrow (\text{Nat} \rightarrow \text{Nat Maybe}) \rightarrow \text{Nat Maybe} \rightarrow \text{Nat Maybe})$
 $\rightarrow \text{Expr} \rightarrow (\text{Nat} \rightarrow \text{Nat Maybe}) \rightarrow \text{Nat Maybe} \rightarrow \text{Nat Maybe}$

where

eval-body-final·*r*·(*Val*·*n*)·*s*·*f* = *s*·*n*
| *eval-body-final*·*r*·(*Add*·*x*·*y*)·*s*·*f* = *r*·*x*·($\Lambda n. r\cdot y\cdot(\Lambda m. s\cdot(n + m))\cdot f$)·*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]*: $\text{eval-body-final}\cdot r\cdot \perp\cdot s\cdot f = \perp$
by (*subst eval-body-final.unfold, simp*)

lemma *eval-body'-eval-body-final-eq*: $\text{eval-body-final} \circ \text{unwrapC} \circ \text{wrapC} = \text{eval-body}'$
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 llist =

lnil
| *lcons* (**lazy** 'a) (**lazy** 'a *llist*)

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

The following standard list functions underpin the monadic infrastructure:

fixrec *lappend* :: 'a *llist* \rightarrow 'a *llist* \rightarrow 'a *llist* **where**
lappend·*lnil*·*ys* = *ys*
| *lappend*·(*lcons*·*x*·*xs*)·*ys* = *lcons*·*x*·(*lappend*·*xs*·*ys*)

fixrec *lconcat* :: 'a *llist* *llist* \rightarrow 'a *llist* **where**
lconcat·*lnil* = *lnil*
| *lconcat*·(*lcons*·*x*·*xs*) = *lappend*·*x*·(*lconcat*·*xs*)

fixrec *lmap* :: ('a \rightarrow 'b) \rightarrow 'a *llist* \rightarrow 'b *llist* **where**
lmap·*f*·*lnil* = *lnil*
| *lmap*·*f*·(*lcons*·*x*·*xs*) = *lcons*·(*f*·*x*)·(*lmap*·*f*·*xs*)

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

type-synonym *S* = *nat discr llist*

definition *returnS* :: *nat discr* \rightarrow *S* **where**
returnS = (Λ *x*. *lcons*·*x*·*lnil*)

definition *bindS* :: *S* \rightarrow (*nat discr* \rightarrow *S*) \rightarrow *S* **where**
bindS = (Λ *x g*. *lconcat*·(*lmap*·*g*·*x*))

Unfortunately the lack of higher-order polymorphism in HOL prevents us from providing the general typing one would expect a monad to have in Haskell.

The evaluator uses the following extra constants:

definition *addS* :: *S* \rightarrow *S* \rightarrow *S* **where**
addS \equiv (Λ *x y*. *bindS*·*x*·(Λ *xv*. *bindS*·*y*·(Λ *yv*. *returnS*·(*xv* + *yv*))))

definition *disjS* :: *S* \rightarrow *S* \rightarrow *S* **where**
disjS \equiv *lappend*

definition *failS* :: *S* **where**
failS \equiv *lnil*

We interpret our language using these combinators in the obvious way. The only complication is that, even though our evaluator is primitive recursive, we must explicitly use the fixed point operator as the worker/wrapper technique requires us to talk about the body of the recursive definition.

definition

$$\begin{aligned} \text{evalS-body} &:: (\text{expr } \text{discr} \rightarrow \text{nat } \text{discr } \text{lList}) \\ &\rightarrow (\text{expr } \text{discr} \rightarrow \text{nat } \text{discr } \text{lList}) \end{aligned}$$
where

$$\begin{aligned} \text{evalS-body} &\equiv \Lambda r e. \text{case } \text{undiscr } e \text{ of} \\ &\quad \text{const } n \Rightarrow \text{returnS} \cdot (\text{Discr } n) \\ &\quad | \text{add } e1 e2 \Rightarrow \text{addS} \cdot (r \cdot (\text{Discr } e1)) \cdot (r \cdot (\text{Discr } e2)) \\ &\quad | \text{disj } e1 e2 \Rightarrow \text{disjS} \cdot (r \cdot (\text{Discr } e1)) \cdot (r \cdot (\text{Discr } e2)) \\ &\quad | \text{fail} \Rightarrow \text{failS} \end{aligned}$$
abbreviation $\text{evalS} :: \text{expr } \text{discr} \rightarrow \text{nat } \text{discr } \text{lList}$ **where**

$$\text{evalS} \equiv \text{fix} \cdot \text{evalS-body}$$

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

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

type-synonym $\text{Obs} = \text{nat } \text{discr } \text{lList}$

type-synonym $\text{Failure} = \text{Obs}$

type-synonym $\text{Success} = \text{nat } \text{discr} \rightarrow \text{Failure} \rightarrow \text{Obs}$

type-synonym $K = \text{Success} \rightarrow \text{Failure} \rightarrow \text{Obs}$

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

The monad over the continuation type K is as follows:

definition $\text{returnK} :: \text{nat } \text{discr} \rightarrow K$ **where**

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

$$\text{bindK} \equiv \Lambda x g. \Lambda s f. x \cdot (\Lambda xv f'. g \cdot xv \cdot s \cdot f') \cdot f$$

Our extra constants are defined as follows:

definition $\text{addK} :: K \rightarrow K \rightarrow K$ **where**

$$\text{addK} \equiv (\Lambda x y. \text{bindK} \cdot x \cdot (\Lambda xv. \text{bindK} \cdot y \cdot (\Lambda yv. \text{returnK} \cdot (xv + yv))))$$
definition $\text{disjK} :: K \rightarrow K \rightarrow K$ **where**

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

$$\text{failK} \equiv \Lambda s f. f$$

The continuation semantics is again straightforward:

definition

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

where

$$\begin{aligned} \text{evalK-body} &\equiv \Lambda r e. \text{case undiscr } e \text{ of} \\ &\quad \text{const } n \Rightarrow \text{returnK} \cdot (\text{Discr } n) \\ &\quad | \text{add } e1 \ e2 \Rightarrow \text{addK} \cdot (r \cdot (\text{Discr } e1)) \cdot (r \cdot (\text{Discr } e2)) \\ &\quad | \text{disj } e1 \ e2 \Rightarrow \text{disjK} \cdot (r \cdot (\text{Discr } e1)) \cdot (r \cdot (\text{Discr } e2)) \\ &\quad | \text{fail} \Rightarrow \text{failK} \end{aligned}$$

abbreviation $\text{evalK} :: \text{expr } \text{discr} \rightarrow K$ **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.

fixrec $SK :: S \rightarrow K$ **where**

$$\begin{aligned} SK \cdot \text{lnil} &= \text{failK} \\ | SK \cdot (\text{lcons} \cdot x \cdot xs) &= (\Lambda s f. s \cdot x \cdot (SK \cdot xs \cdot s \cdot f)) \end{aligned}$$

definition

$$\text{unwrap} :: (\text{expr } \text{discr} \rightarrow \text{nat } \text{discr } \text{llist}) \rightarrow (\text{expr } \text{discr} \rightarrow K)$$

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 $KS :: K \rightarrow S$ **where**

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

definition $\text{wrap} :: (\text{expr } \text{discr} \rightarrow K) \rightarrow (\text{expr } \text{discr} \rightarrow \text{nat } \text{discr } \text{llist})$ **where**

$$\text{wrap} \equiv \Lambda r e. KS \cdot (r \cdot e)$$

The worker/wrapper condition follows directly from these definitions.

lemma $KS \cdot SK \cdot \text{id}$:

$$\begin{aligned} KS \cdot (SK \cdot xs) &= xs \\ \text{by } (\text{induct } xs) \text{ (simp-all add: KS-def failK-def)} \end{aligned}$$

lemma $\text{wrap} \cdot \text{unwrap} \cdot \text{id}$:

$$\begin{aligned} \text{wrap } oo \ \text{unwrap} &= ID \\ \text{unfolding } \text{wrap-def } \text{unwrap-def} \\ \text{by } (\text{simp add: KS-SK-id cfun-eq-iff}) \end{aligned}$$

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 :

$$\begin{aligned} f \cdot x \not\sqsubseteq g \cdot x &\implies f \not\sqsubseteq g \\ \text{by } (\text{auto simp: cfun-below-iff}) \end{aligned}$$

lemma $\text{unwrap} \cdot \text{wrap} \cdot \text{not-under-id}$:

$unwrap \circ wrap \sqsubseteq ID$
proof –
let $?witness = \Lambda e. (\Lambda s f. lnil :: K)$
have $(unwrap \circ wrap) \cdot ?witness \cdot (Discr\ fail) \cdot \perp \cdot (lcons \cdot 0 \cdot lnil)$
 $\sqsubseteq ?witness \cdot (Discr\ fail) \cdot \perp \cdot (lcons \cdot 0 \cdot lnil)$
by $(simp\ add: failK-def\ wrap-def\ unwrap-def\ KS-def)$
hence $(unwrap \circ wrap) \cdot ?witness \sqsubseteq ?witness$
by $(fastforce\ intro!: cfun-not-below)$
thus $?thesis$ **by** $(simp\ add: cfun-not-below)$
qed

We now apply `worker_wrapper_id`:

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

definition $eval-wv :: expr\ discr \rightarrow nat\ discr\ llist$ **where**
 $eval-wv \equiv wrap \cdot eval-work$

lemma $evalS = eval-wv$
unfolding $eval-wv-def\ eval-work-def$
using $worker-wrapper-id[OF\ wrap-unwrap-id]$
by $simp$

We now show how the monadic operations correspond by showing that SK witnesses a *monad morphism* (Wadler 1992, §6). As required by Danvy et al. (2001, Definition 2.1), the mapping needs to hold for our specific operations in addition to the common monadic scaffolding.

lemma $SK-returnS-returnK$:
 $SK \cdot (returnS \cdot x) = returnK \cdot x$
by $(simp\ add: returnS-def\ returnK-def\ failK-def)$

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

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

lemma $SK-addS-distrib$:
 $SK \cdot (addS \cdot x \cdot y) = addK \cdot (SK \cdot x) \cdot (SK \cdot y)$
by $(clarsimp\ simp: ccomp1$
 $addS-def\ addK-def\ failK-def$
 $SK-bindS-bindK\ SK-returnS-returnK)$

lemma $SK-disjS-disjK$:

$SK \cdot (disjS \cdot xs \cdot ys) = disjK \cdot (SK \cdot xs) \cdot (SK \cdot ys)$
by (*simp add: cfun-eq-iff disjS-def disjK-def SK-lappend-distrib*)

lemma *SK-failS-failK*:
 $SK \cdot failS = failK$
unfolding *failS-def* **by** *simp*

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

lemma *evalS-body-evalK-body*:
 $unwrap \circ evalS\text{-body} \circ wrap = evalK\text{-body} \circ unwrap \circ wrap$
proof(*intro cfun-eqI*)
fix $r \ e' \ s \ f$
obtain $e :: expr$
where $ee' : e' = Discr \ e$ **by** (*cases e'*)
have $(unwrap \circ evalS\text{-body} \circ wrap) \cdot r \cdot (Discr \ e) \cdot s \cdot f$
 $= (evalK\text{-body} \circ unwrap \circ wrap) \cdot r \cdot (Discr \ e) \cdot s \cdot f$
by (*cases e*)
(*simp-all add: evalS-body-def evalK-body-def unwrap-def*
 $SK\text{-returnS-returnK} \ SK\text{-addS-distrib}$
 $SK\text{-disjS-disjK} \ SK\text{-failS-failK}$)
with ee' **show** $(unwrap \circ evalS\text{-body} \circ wrap) \cdot r \cdot e' \cdot s \cdot f$
 $= (evalK\text{-body} \circ unwrap \circ wrap) \cdot r \cdot e' \cdot s \cdot f$
by *simp*
qed

theorem *evalS-evalK*:
 $evalS = wrap \cdot evalK$
using *worker-wrapper-fusion-new[OF wrap-unwrap-id unwrap-strict]*
 $evalS\text{-body-evalK-body}$
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 $O(n \lg n)$ one

Andy Gill's solution, mechanised.

10.1 The *nub* function.

fixrec *nub* :: *Nat llist* → *Nat llist*

where

nub.nil = *lnil*

| *nub*·(*x* :@ *xs*) = *x* :@ *nub*·(*lfilter*·(*neg oo* (Λ *y*. *x* =_B *y*))·*xs*)

lemma *nub-strict[simp]*: *nub*·⊥ = ⊥

by *fixrec-simp*

fixrec *nub-body* :: (*Nat llist* → *Nat llist*) → *Nat llist* → *Nat llist*

where

nub-body.f.nil = *lnil*

| *nub-body.f*·(*x* :@ *xs*) = *x* :@ *f*·(*lfilter*·(*neg oo* (Λ *y*. *x* =_B *y*))·*xs*)

lemma *nub-nub-body-eg*: *nub* = *fix.nub-body*

by (*rule cfun-egI*, *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 *NatSet* = *Nat llist*

definition

SetEmpty :: *NatSet* **where**

SetEmpty ≡ *lnil*

definition

SetInsert :: *Nat* → *NatSet* → *NatSet* **where**

SetInsert ≡ *lcons*

definition

SetMem :: *Nat* → *NatSet* → *tr* **where**

SetMem ≡ *lmember*·(*bpred* (=))

lemma *SetMem-strict[simp]*: *SetMem*·*x*·⊥ = ⊥ **by** (*simp add: SetMem-def*)

lemma *SetMem-SetEmpty[simp]*: *SetMem*·*x*·*SetEmpty* = *FF*

by (*simp add: SetMem-def SetEmpty-def*)

lemma *SetMem-SetInsert*: *SetMem*·*v*·(*SetInsert*·*x*·*s*) = (*SetMem*·*v*·*s* *orelse* *x* =_B *v*)

by (*simp add: SetMem-def SetInsert-def*)

AndyG's new type.

domain *R* = *R* (**lazy** *resultR* :: *Nat llist*) (**lazy** *exceptR* :: *NatSet*)

definition

nextR :: *R* → (*Nat* * *R*) *Maybe* **where**

nextR = (Λ *r*. *case ldropWhile*·(Λ *x*. *SetMem*·*x*·(*exceptR*·*r*))·(*resultR*·*r*) *of*

$$\begin{aligned} & \text{nil} \Rightarrow \text{Nothing} \\ & | x :@ xs \Rightarrow \text{Just} \cdot (x, R \cdot xs \cdot (\text{exceptR} \cdot r)) \end{aligned}$$

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

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

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

definition

$$\begin{aligned} & \text{filterR} :: \text{Nat} \rightarrow R \rightarrow R \text{ where} \\ & \text{filterR} \equiv (\Lambda v r. R \cdot (\text{resultR} \cdot r) \cdot (\text{SetInsert} \cdot v \cdot (\text{exceptR} \cdot r))) \end{aligned}$$

definition

$$\begin{aligned} & c2a :: \text{Nat llist} \rightarrow R \text{ where} \\ & c2a \equiv \Lambda xs. R \cdot xs \cdot \text{SetEmpty} \end{aligned}$$

definition

$$\begin{aligned} & a2c :: R \rightarrow \text{Nat llist} \text{ where} \\ & a2c \equiv \Lambda r. \text{lfilter} \cdot (\Lambda v. \text{neg} \cdot (\text{SetMem} \cdot v \cdot (\text{exceptR} \cdot r))) \cdot (\text{resultR} \cdot r) \end{aligned}$$

lemma *a2c-strict*[simp]: $a2c \cdot \perp = \perp$ **unfolding** *a2c-def* **by** *simp*

lemma *a2c-c2a-id*: $a2c \circ c2a = \text{ID}$

by (*rule cfun-eqI, simp add: a2c-def c2a-def lfilter-const-true*)

definition

$$\begin{aligned} & \text{wrap} :: (R \rightarrow \text{Nat llist}) \rightarrow \text{Nat llist} \rightarrow \text{Nat llist} \text{ where} \\ & \text{wrap} \equiv \Lambda f xs. f \cdot (c2a \cdot xs) \end{aligned}$$

definition

$$\begin{aligned} & \text{unwrap} :: (\text{Nat llist} \rightarrow \text{Nat llist}) \rightarrow R \rightarrow \text{Nat llist} \text{ where} \\ & \text{unwrap} \equiv \Lambda f r. f \cdot (a2c \cdot r) \end{aligned}$$

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

unfolding *unwrap-def* **by** (*rule cfun-eqI, simp*)

lemma *wrap-unwrap-id*: $\text{wrap} \circ \text{unwrap} = \text{ID}$

using *cfun-fun-cong[OF a2c-c2a-id]*

by $-$ (*rule cfun-eqI*), *simp add: wrap-def unwrap-def*)

Equivalences needed for later.

lemma *TR-deMorgan*: $\text{neg} \cdot (x \text{ or else } y) = (\text{neg} \cdot x \text{ and also } \text{neg} \cdot y)$

by (*rule trE[where p=x], simp-all*)

lemma *case-maybe-case*:

$$\begin{aligned} & (\text{case } (\text{case } L \text{ of } \text{nil} \Rightarrow \text{Nothing} \mid x :@ xs \Rightarrow \text{Just} \cdot (h \cdot x \cdot xs)) \text{ of} \\ & \quad \text{Nothing} \Rightarrow f \mid \text{Just} \cdot (a, b) \Rightarrow g \cdot a \cdot b) \end{aligned}$$

$=$

$$(\text{case } L \text{ of } \text{nil} \Rightarrow f \mid x :@ xs \Rightarrow g \cdot (\text{fst } (h \cdot x \cdot xs)) \cdot (\text{snd } (h \cdot x \cdot xs)))$$

```

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

```

lemma case-a2c-case-caseR:

```

  (case a2c.w of lnil  $\Rightarrow$  f | x :@ xs  $\Rightarrow$  g.x.xs)
= (case nextR.w of Nothing  $\Rightarrow$  f | Just.(x, r)  $\Rightarrow$  g.x.(a2c.r)) (is ?lhs = ?rhs)

```

proof –

```

have ?rhs = (case (case ldropWhile.( $\Lambda$  x. SetMem.x.(exceptR.w)).(resultR.w) of
  lnil  $\Rightarrow$  Nothing
  | x :@ xs  $\Rightarrow$  Just.(x, R.xs.(exceptR.w))) of Nothing  $\Rightarrow$  f | Just.(x,
r)  $\Rightarrow$  g.x.(a2c.r))
  by (simp add: nextR-def)
also have ... = (case ldropWhile.( $\Lambda$  x. SetMem.x.(exceptR.w)).(resultR.w) of
  lnil  $\Rightarrow$  f | x :@ xs  $\Rightarrow$  g.x.(a2c.(R.xs.(exceptR.w))))
  using case-maybe-case[where L=ldropWhile.( $\Lambda$  x. SetMem.x.(exceptR.w)).(resultR.w)
  and f=f and g= $\Lambda$  x r. g.x.(a2c.r) and h= $\Lambda$  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 llist)

apply simp-all

apply (rule-tac p=SetMem.aa.(exceptR.w) **in** trE)

apply simp-all

done

finally show ?lhs = ?rhs **by** simp

qed

lemma filter-filterR: lfilter.(neg oo (Λ y. x =_B y)).(a2c.r) = a2c.(filterR.x.r)

using filter-filter[**where** p=Tr.neg oo (Λ y. x =_B y) **and** q= Λ v. Tr.neg.(SetMem.v.(exceptR.r))]

unfolding a2c-def filterR-def

by (cases r, simp-all add: SetMem-SetInsert TR-deMorgan)

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

definition

nub-body' :: (R → Nat llist) → R → Nat llist **where**

nub-body' \equiv Λ f r. case a2c.r of lnil \Rightarrow lnil

| x :@ xs \Rightarrow 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.($\Lambda v. Tr.neg.(SetMem.v.(exceptR.xa))$).(resultR.xa)
, simp-all add: fix-const)

definition

nub-body'' :: (R → Nat llist) → R → Nat llist **where**
nub-body'' ≡ $\Lambda f r. case nextR.r of Nothing \Rightarrow lnil$
| Just.(x, xs) ⇒ x :@ f.(c2a.(lfilter.(neg oo ($\Lambda y. x$
=ₐ 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= $\Lambda x xs. x :@ f.(c2a.(lfilter.(Tr.neg$
oo ($\Lambda y. x =_B y$)).xs)) **and** w=r]

by simp

qed

definition

nub-body''' :: (R → Nat llist) → R → Nat llist **where**
nub-body''' ≡ ($\Lambda f r. case nextR.r of Nothing \Rightarrow lnil$
| Just.(x, xs) ⇒ x :@ f.(filterR.x.xs))

lemma nub-body''-nub-body'''-eq: nub-body'' = nub-body''' oo (unwrap oo wrap)

unfolding nub-body''-def nub-body'''-def wrap-def unwrap-def

by ((rule cfun-eqI)+, simp add: filter-filterR)

Finally glue it all together.

lemma nub-wrap-nub-body''': nub = wrap.(fix.nub-body''')

using worker-wrapper-fusion-new[OF wrap-unwrap-id unwrap-strict, **where** body=nub-body]

nub-nub-body-eq

nub-body-nub-body'-eq

nub-body'-nub-body''-eq

nub-body''-nub-body'''-eq

by simp

end

11 Optimise “last”.

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

11.1 The last function.

fixrec llast :: 'a llist → 'a

where

llast.(x :@ yys) = (case yys of lnil ⇒ x | y :@ ys ⇒ llast.yys)

lemma *llast-strict[simp]*: $llast.\perp = \perp$

by *fixrec-simp*

fixrec *llast-body* :: $('a\ list \rightarrow 'a) \rightarrow 'a\ list \rightarrow 'a$

where

$llast-body.f(x :@ yys) = (case\ yys\ of\ lnil \Rightarrow x \mid y :@ ys \Rightarrow f.yys)$

lemma *llast-llast-body*: $llast = fix.llast-body$

by (*rule cfun-eqI, subst llast-def, subst llast-body.unfold, simp*)

definition *wrap* :: $('a \rightarrow 'a\ list \rightarrow 'a) \rightarrow ('a\ list \rightarrow 'a)$ **where**

$wrap \equiv \Lambda f\ (x :@ xs). f.x.xs$

definition *unwrap* :: $('a\ list \rightarrow 'a) \rightarrow ('a \rightarrow 'a\ list \rightarrow 'a)$ **where**

$unwrap \equiv \Lambda f\ x\ xs. f.(x :@ xs)$

lemma *unwrap-strict[simp]*: $unwrap.\perp = \perp$

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* :: $('a \rightarrow 'a\ list \rightarrow 'a) \rightarrow 'a \rightarrow 'a\ list \rightarrow 'a$ **where**

$llast-worker \equiv \Lambda r\ x\ yys. case\ yys\ of\ lnil \Rightarrow x \mid y :@ ys \Rightarrow r.y.yys$

definition *llast'* :: $'a\ list \rightarrow 'a$ **where**

$llast' \equiv wrap.(fix.llast-worker)$

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

unfolding *llast-worker-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** *?lhs = ?rhs*)

proof –

have *?rhs = fix.llast-body* **by** (*simp only: llast-llast-body*)

also have $\dots = wrap.(fix.(unwrap\ oo\ llast-body\ oo\ wrap))$

by (*simp only: worker-wrapper-body[OF wrap-unwrap-ID]*)

also have $\dots = wrap.(fix.(llast-worker))$

by (*simp only: llast-worker-llast-body*)

also have $\dots = ?lhs$ **unfolding** *llast'-def* **by** *simp*

finally show *?thesis* **by** *simp*

qed

end

12 Concluding remarks

Gill and Hutton provide two examples of fusion: accumulator introduction in their §4, and the transformation in their §7 of an interpreter for a language with exceptions into one employing continuations. Both involve strict *unwraps* and are indeed totally correct.

The example in their §5 demonstrates the unboxing of numerical computations using a different worker/wrapper rule and does not require fusion. In their §6 a non-strict *unwrap* is used to memoise functions over the natural numbers using the rule considered here. It should in fact use the same rule as the unboxing example as the scheme only correctly memoises strict functions. We can see this by considering a base case missing from their inductive proof, viz that if $f :: Nat \rightarrow a$ is not strict – in fact constant, as Nat is a flat domain – then $f \perp \neq \perp = (map\ f\ [0..]) !! \perp$, where $xs !! n$ is the n th element of xs .

References

- H. Bekić. Definable operation in general algebras, and the theory of automata and flowcharts. In *Programming Languages and Their Definition - Hans Bekić (1936-1982)*, pages 30–55. Springer-Verlag, 1984.
- R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *J. ACM*, 24(1):44–67, 1977. doi: <http://doi.acm.org/10.1145/321992.321996>.
- O. Danvy, B. Grobauer, and M. Rhiger. A unifying approach to goal-directed evaluation. *New Generation Comput.*, 20(1):53–74, 2001.
- J. W. de Bakker, A. de Bruin, and J. Zucker. *Mathematical theory of program correctness*. Prentice-Hall international series in computer science. Prentice Hall, 1980.
- W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and their Comparison*. Cambridge University Press, 1998.
- M. M. Fokkinga and Erik Meijer. Program Calculation Properties of Continuous Algebras. Technical Report CS-R9104, CWI, 1991.
- P. Gammie. Short note: Strict unwraps make worker/wrapper fusion totally correct. *Journal of Functional Programming*, 21:209–213, 2011.

- A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.
- S. Greibach. *Theory of program structures: schemes, semantics, verification*, volume 36 of *LNCS*. Springer-Verlag, 1975.
- D. Harel. On folk theorems. *C. ACM*, 23(7):379–389, 1980. doi: <http://doi.acm.org/10.1145/358886.358892>.
- B. Huffman. A purely definitional universal domain. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *TPHOLs*, volume 5674 of *LNCS*, pages 260–275. Springer, 2009. ISBN 978-3-642-03358-2.
- G. Hutton, M. Jaskelioff, and A. Gill. Factorising folds for faster functions. *Journal of Functional Programming*, 20(3-4):353–373, 2010.
- Z. Manna. *Introduction to Mathematical Theory of Computation*. McGraw-Hill, Inc., New York, NY, USA, 1974. ISBN 0070399107.
- E. Meijer, M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 124–144. Springer-Verlag, 1991.
- O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.
- S. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM Conference on Functional programming languages and computer architecture*, pages 636–666. Springer-Verlag, 1991.
- A. M. Pitts. Relational properties of domains. *Information and Computation*, 127:66–90, 1996.
- G. Plotkin. Pisa Notes (on Domain Theory). Unpublished, 1983. URL <http://homepages.inf.ed.ac.uk/gdp/publications/Domains.ps>.
- J. C. Reynolds. On the relation between direct and continuation semantics. In J. Loeckx, editor, *ICALP*, volume 14 of *LNCS*, pages 141–156. Springer, 1974.
- Davide Sangiorgi. On the origins of bisimulation and coinduction. *ACM Trans. Program. Lang. Syst.*, 31(4), 2009.
- A. K. Simpson and G. D. Plotkin. Complete axioms for categorical fixed-point operators. In *LICS*, pages 30–41, 2000.

- J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.
- M. Tullsen. *PATH, a Program Transformation System for Haskell*. PhD thesis, Yale University, New Haven, CT, USA, 2002. URL <http://www.cs.yale.edu/publications/techreports/tr1229.pdf>.
- P. Wadler. Comprehending monads. *MSCS*, 2:461–493, 1992.
- M. Wand and D. Vaillancourt. Relating models of backtracking. In C. Okasaki and K. Fisher, editors, *ICFP*, pages 54–65. ACM, 2004. ISBN 1-58113-905-5.
- G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.