Mechanising the worker/wrapper transformation

Peter Gammie

March 17, 2025

Contents

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.
$\langle ML \rangle$

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).

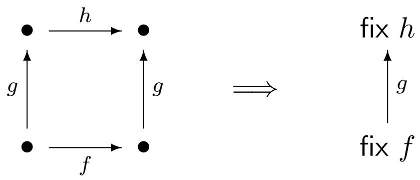
lemma *rolling-rule-ltr*: $fix\cdot(g\ oo\ f) \sqsubseteq g\cdot(fix\cdot(f\ oo\ g))$
$\langle proof \rangle$

lemma *rolling-rule-rtl*: $g\cdot(fix\cdot(f\ oo\ g)) \sqsubseteq fix\cdot(g\ oo\ f)$
$\langle proof \rangle$

lemma *rolling-rule*: $fix\cdot(g\ oo\ f) = g\cdot(fix\cdot(f\ oo\ g))$
$\langle proof \rangle$

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`:

$$\frac{\mathit{adm}\ (\lambda x.\ ?P\ (\mathit{fst}\ x)\ (\mathit{snd}\ x)) \qquad ?P\ \bot\ \bot \qquad \bigwedge x\ y.\ \dfrac{?P\ x\ y}{?P\ (?F{\cdot}x)\ (?G{\cdot}y)}}{?P\ (\mathit{fix}{\cdot}?F)\ (\mathit{fix}{\cdot}?G)}$$

in a very straightforward way:

lemma *lfp-fusion*:
  assumes $g{\cdot}\bot = \bot$
  assumes $g\ oo\ f = h\ oo\ g$
  shows $g{\cdot}(\mathit{fix}{\cdot}f) = \mathit{fix}{\cdot}h$
⟨*proof*⟩

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

⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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.

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

$$comp = wrap\ work$$
$$work :: B \qquad \text{(the worker/wrapper}$$
$$work = \mathsf{fix}\ (unwrap \circ body \circ wrap)$$

transformation)

Also:

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

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

The first requires *wrap oo unwrap* to be the identity for all values.

lemma *worker-wrapper-id*:
  fixes *wrap* :: *'b::pcpo → 'a::pcpo*
  fixes *unwrap* :: *'a → 'b*
  assumes *wrap-unwrap*: *wrap oo unwrap = ID*
  assumes *comp-body*: *computation = fix·body*
  shows *computation = wrap·(fix·(unwrap oo body oo wrap))*
⟨*proof*⟩

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*⟩

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*⟩

Gill and Hutton's *worker-wrapper-fusion* rule is intended to allow the transformation of $(\mathit{unwrap}\ \mathit{oo}\ \mathit{wrap}){\cdot}R$ 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 $\mathit{wrap} :: {}'b{::}\mathit{pcpo} \to {}'a{::}\mathit{pcpo}$
  fixes $\mathit{unwrap} :: {}'a \to {}'b$
  assumes *wrap-unwrap*: $\mathit{wrap}\ \mathit{oo}\ \mathit{unwrap} = \mathit{ID}$
  assumes *work*: $\mathit{work} = \mathit{fix}{\cdot}(\mathit{unwrap}\ \mathit{oo}\ \mathit{body}\ \mathit{oo}\ \mathit{wrap})$
  shows $(\mathit{unwrap}\ \mathit{oo}\ \mathit{wrap}){\cdot}\mathit{work} = \mathit{work}$
⟨*proof*⟩

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 $f = \mathit{body}$ yields a rule of the form $\mathit{body} \implies f$, one can always derive $f = f$. Clearly this has dire implications for the preservation of termination behaviour.

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

$$\frac{f{\cdot}?x = ?x}{\mathit{fix}{\cdot}f \sqsubseteq ?x}\ \texttt{fix\_least}$$

viz that $f\ x = x$ implies only the partially correct $\mathit{fix}\ f \sqsubseteq x$, and not the totally correct $\mathit{fix}\ f = x$. We use this characterisation to show that if *unwrap* is non-strict (i.e. $\mathit{unwrap}\ \bot \neq \bot$) then there are programs where worker/wrapper fusion as used by Gill and Hutton need only be partially correct.

Consider the scenario described in Figure 1. After applying the worker/wrapper transformation, we attempt to apply fusion by finding a residual expression *body′* such that the body of the worker, i.e. the expression *unwrap oo body oo wrap*, can be rewritten as *body′ oo unwrap oo wrap*. Intuitively this is the semantic form of workers where all self-calls are fusible. Our goal is to justify redefining *work* to *fix·body′*, i.e. to establish:

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

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*: $\mathit{wrap}\ \mathit{oo}\ \mathit{unwrap} = \mathit{ID}$
  assumes *work*: $\mathit{work} = \mathit{fix}\cdot(\mathit{unwrap}\ \mathit{oo}\ \mathit{body}\ \mathit{oo}\ \mathit{wrap})$
  assumes *body′*: $\mathit{unwrap}\ \mathit{oo}\ \mathit{body}\ \mathit{oo}\ \mathit{wrap} = \mathit{body}'\ \mathit{oo}\ \mathit{unwrap}\ \mathit{oo}\ \mathit{wrap}$
  shows $\mathit{fix}\cdot\mathit{body}' \sqsubseteq \mathit{work}$
⟨*proof*⟩

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 $\forall\, x.\ ?f\cdot(?g\cdot x) = x \Longrightarrow ?f\cdot\bot = \bot$.

fixrec $\mathit{wrap} :: B \to A$
where $\mathit{wrap}\cdot(B\cdot a) = a$
⟨*proof*⟩
fixrec $\mathit{unwrap} :: A \to B$
where $\mathit{unwrap} = B$

Discharging the worker/wrapper hypothesis is similarly routine.

lemma *wrap-unwrap*: $\mathit{wrap}\ \mathit{oo}\ \mathit{unwrap} = \mathit{ID}$
  ⟨*proof*⟩

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

fixrec $\mathit{body} :: A \to A$
where $\mathit{body}\cdot r = A$

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

fixrec $body' :: B \to B$
where $body' \cdot (B \cdot a) = B \cdot A$⟨*proof*⟩

As explained above, we set up the fusion opportunity:

lemma *body-body'*: $unwrap\ oo\ body\ oo\ wrap = body'\ oo\ unwrap\ oo\ wrap$
⟨*proof*⟩

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

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

lemma $fix \cdot body' \sqsubseteq fix \cdot (unwrap\ oo\ body\ oo\ wrap)$
⟨*proof*⟩

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

lemma $\neg\ fix \cdot (unwrap\ oo\ body\ oo\ wrap) \sqsubseteq fix \cdot body'$
⟨*proof*⟩

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

lemma
assumes $wrap\ oo\ unwrap = ID$
shows $wrap\ oo\ strictify \cdot unwrap = ID$
⟨*proof*⟩

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.

⟨*proof*⟩

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 $wrap\ oo\ unwrap = ID$.)

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*:

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

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

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

$$unwrap\ (wrap\ work) \Longrightarrow work$$ (worker/wrapper fusion)

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

fixes $wrap :: {}'b{::}pcpo \to {}'a{::}pcpo$
fixes $unwrap :: {}'a \to {}'b$
fixes $body' :: {}'b \to {}'b$
assumes *wrap-unwrap*: $wrap\ oo\ unwrap = (ID :: {}'a \to {}'a)$
assumes *unwrap-strict*: $unwrap{\cdot}\bot = \bot$
assumes *body-body'*: $unwrap\ oo\ body\ oo\ wrap = body'\ oo\ (unwrap\ oo\ wrap)$
shows $fix{\cdot}body = wrap{\cdot}(fix{\cdot}body')$
⟨*proof*⟩

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 $wrap :: {}'b{::}pcpo \to {}'a{::}pcpo$
fixes $unwrap :: {}'a \to {}'b$
assumes *wrap-unwrap*: $wrap\ oo\ unwrap = (ID :: {}'a \to {}'a)$
assumes *unwrap-strict*: $unwrap{\cdot}\bot = \bot$
assumes *body-body'*: $\bigwedge r.\ (unwrap\ oo\ wrap){\cdot}r = r$
$\Longrightarrow (unwrap\ oo\ body\ oo\ wrap){\cdot}r = body'{\cdot}r$
shows $fix{\cdot}body = wrap{\cdot}(fix{\cdot}body')$
⟨*proof*⟩

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* → *'a llist*

definition
list2H :: *'a llist* → *'a H* **where**
list2H ≡ *lappend*

lemma *acc-c2a-strict*[*simp*]: *list2H*·⊥ = ⊥
⟨*proof*⟩

definition
H2list :: *'a H* → *'a llist* **where**
H2list ≡ Λ *f* . *f*·*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 ⊥ = (Λ *x*. ⊥), which does not hold in Haskell.

lemma *H-llist-hom-append*: *list2H*·(*xs* :++ *ys*) = *list2H*·*xs* *oo* *list2H*·*ys* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *H-llist-hom-id*: *list2H*·*lnil* = *ID* ⟨*proof*⟩

lemma *H2list-list2H-inv*: *H2list* *oo* *list2H* = *ID*
⟨*proof*⟩

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

fixrec *lrev* :: *'a llist* → *'a llist*
where
lrev·*lnil* = *lnil*
| *lrev*·(*x* :@ *xs*) = *lrev*·*xs* :++ (*x* :@ *lnil*)

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

lemma *lrev-strict*[*simp*]: *lrev*·⊥ = ⊥
⟨*proof*⟩

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

lemma *lrev-body-strict*[*simp*]: *lrev-body*·*r*·⊥ = ⊥

⟨*proof*⟩

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*
⟨*proof*⟩

Wrap / unwrap functions.

definition
unwrapH :: (′*a llist* → ′*a llist*) → ′*a llist* → ′*a H* **where**
unwrapH ≡ Λ *f xs* . *list2H*·(*f*·*xs*)

lemma *unwrapH-strict*[*simp*]: *unwrapH*·⊥ = ⊥
⟨*proof*⟩

definition
wrapH :: (′*a llist* → ′*a H*) → ′*a llist* → ′*a llist* **where**
wrapH ≡ Λ *f xs* . *H2list*·(*f*·*xs*)

lemma *wrapH-unwrapH-id*: *wrapH oo unwrapH* = *ID* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

5.2 Gill/Hutton-style worker/wrapper.

definition
lrev-work :: ′*a llist* → ′*a H* **where**
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*
⟨*proof*⟩

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*

⟨*proof*⟩

lemma *lrev-work1-lrev-work-eq*: *lrev-work1* = *lrev-work*
⟨*proof*⟩

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·r·*⊥ = ⊥
⟨*proof*⟩

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

lemma *lrev-work2-strict*[*simp*]: *lrev-work2·*⊥ = ⊥
⟨*proof*⟩

lemma *lrev-body2-lrev-body1-eq*: *lrev-body2* = *lrev-body1*
⟨*proof*⟩

lemma *lrev-work2-lrev-work1-eq*: *lrev-work2* = *lrev-work1*
⟨*proof*⟩

Simplify.

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

lemma *lrev-body3-strict*[*simp*]: *lrev-body3·r·*⊥ = ⊥
⟨*proof*⟩

definition
lrev-work3 :: $'a$ *llist* → $'a$ *H* **where**
lrev-work3 ≡ *fix·lrev-body3*

lemma *lrev-wwfusion*: *list2H·*((*wrapH·lrev-work2*)*·xs*) = *lrev-work2·xs*
⟨*proof*⟩

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

lemma *lrev-work3* ⊑ *lrev-work2*
⟨*proof*⟩

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 *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.

lemma *lrev-work3-lrev-work2-eq*: *lrev-work3* = *lrev-work2* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lrev-work3-lrev-work2-eq'*: *lrev* = *wrapH·lrev-work3*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *lrev-body-final-lrev-body3-eq*: *lrev-body-final* = *lrev-body3*
⟨*proof*⟩

lemma *lrev-final-lrev-eq*: *lrev* = *lrev-final* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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,

$UNat_\bot$ the lifted (flat and pointed) variant, and *Nat* the standard boxed domain, isomorphic to $UNat_\bot$. 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
  $fac \cdot n = If\ n =_B 0\ then\ 1\ else\ n * fac \cdot (n - 1)$

declare *fac.simps*[*simp del*]

lemma *fac-strict*[*simp*]: $fac \cdot \bot = \bot$
⟨*proof*⟩

definition
  fac-body :: (*Nat* → *Nat*) → *Nat* → *Nat* **where**
  $fac\text{-}body \equiv \Lambda\ r\ n.\ If\ n =_B 0\ then\ 1\ else\ n * r \cdot (n - 1)$

lemma *fac-body-strict*[*simp*]: $fac\text{-}body \cdot r \cdot \bot = \bot$
  ⟨*proof*⟩

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

Wrap / unwrap functions. Note the explicit lifting of the co-domain. For some reason the published version of Gill and Hutton (2009) does not discuss this point: if we're going to handle recursive functions, we need a bottom.

unbox simply removes the tag, yielding a possibly-divergent unboxed value, the result of the function.

definition
  unwrapB :: (*Nat* → *Nat*) → *UNat* → $UNat_\bot$ **where**
  $unwrapB \equiv \Lambda\ f.\ unbox\ oo\ f\ oo\ box$

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

definition
  wrapB :: (*UNat* → $UNat_\bot$) → *Nat* → *Nat* **where**
  $wrapB \equiv \Lambda\ f\ x\ .\ unbox \cdot x >>= f >>= box$

lemma *wrapB-unwrapB-body*:
  assumes *strictF*: $f \cdot \bot = \bot$

shows (*wrapB oo unwrapB*)·*f* = *f* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

Apply worker/wrapper.

definition
fac-work :: *UNat* → $UNat_\bot$ **where**
fac-work ≡ *fix*·(*unwrapB oo fac-body oo wrapB*)

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

lemma *fac-fac-ww-eq*: *fac* = *fac-wrap* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

definition
fac-body′ :: (*UNat* → $UNat_\bot$) → *UNat* → $UNat_\bot$ **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*⟩

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

definition
fac-body-final :: (*UNat* → $UNat_\bot$) → *UNat* → $UNat_\bot$ **where**
fac-body-final ≡ Λ *r n*.
if n = *0 then up*·*1 else r*·(*n* $-_\#$ *1*) >>= (Λ *b*. *up*·(*n* $*_\#$ *b*))

lemma *fac-body-final-fac-body′*: *fac-body-final* = *fac-body′* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

definition
fac-work-final :: *UNat* → $UNat_\bot$ **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*⟩

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 $UNatAcc = UNat \to UNat_\bot$

definition
$n2a :: UNat \to UNatAcc$ **where**
$n2a \equiv \Lambda\ m\ n.\ up \cdot (m *_\#\ n)$

definition
$a2n :: UNatAcc \to UNat_\bot$ **where**
$a2n \equiv \Lambda\ a.\ a \cdot 1$

lemma *a2n-strict*[*simp*]: $a2n \cdot \bot = \bot$
⟨*proof*⟩

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

lemma *A-hom-mult*: $n2a \cdot (x *_\#\ y) = (n2a \cdot x >=> n2a \cdot y)$
⟨*proof*⟩

definition
$unwrapA :: (UNat \to UNat_\bot) \to UNat \to UNatAcc$ **where**
$unwrapA \equiv \Lambda\ f\ n.\ f \cdot n >>= n2a$

lemma *unwrapA-strict*[*simp*]: $unwrapA \cdot \bot = \bot$
⟨*proof*⟩

definition
$wrapA :: (UNat \to UNatAcc) \to UNat \to UNat_\bot$ **where**
$wrapA \equiv \Lambda\ f.\ a2n\ oo\ f$

lemma *wrapA-unwrapA-id*: $wrapA\ oo\ unwrapA = ID$
⟨*proof*⟩

Some steps along the way.

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

lemma *fac-acc-body1-fac-body-final-eq*: *fac-acc-body1* = *unwrapA oo fac-body-final oo wrapA*
⟨*proof*⟩

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*
⟨*proof*⟩

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*⟩

lemma *fac-work-final-body3-eq*: *fac-work-final* = *wrapA·*(*fix·fac-acc-body3*)
⟨*proof*⟩

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* → $UNat_{\bot}$ **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*)
⟨*proof*⟩

lemma *fac-acc-work-final-fac-work*: *fac-acc-work-final* = *fac-work-final* (**is** *?lhs*=*?rhs*)
⟨*proof*⟩

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*)
⟨*proof*⟩
fixrec *smap* :: (*'a* → *'b*) → *'a Stream* → *'b Stream*
where
smap·f·(*x* && *xs*) = *f·x* && *smap·f·xs*
⟨*proof*⟩
lemma *smap-smap*: *smap·f·*(*smap·g·xs*) = *smap·*(*f oo g*)*·xs*⟨*proof*⟩
fixrec *i-th* :: *'a Stream* → *Nat* → *'a*
where
i-th·(*x* && *xs*) = *Nat-case·x·*(*i-th·xs*)

abbreviation
i-th-syn :: *'a Stream* ⇒ *Nat* ⇒ *'a* (**infixl** ‹!!› *100*) **where**
s !! *i* ≡ *i-th·s·i*
⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

The infinite stream of natural numbers.

fixrec *nats* :: *Nat Stream*
where
nats = *0* && *smap·*(Λ *x*. *1* + *x*)*·nats*

7.2 The wrapper/unwrapper functions.

definition
unwrapS' :: (*Nat* → *'a*) → *'a Stream* **where**
unwrapS' ≡ Λ *f* . *smap·f·nats*

lemma *unwrapS'-unfold*: *unwrapS'·f* = *f·0* && *smap·*(*f oo* (Λ *x*. *1* + *x*))*·nats*⟨*proof*⟩
fixrec *unwrapS* :: (*Nat* → *'a*) → *'a Stream*
where
unwrapS·f = *f·0* && *unwrapS·*(*f oo* (Λ *x*. *1* + *x*))

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

lemma *unwrapS-unwrapS'-eq*: *unwrapS* = *unwrapS'* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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*: $(f{::}Nat \to 'a)\cdot\bot = \bot$
  shows $unwrapS\cdot f \,!!\, n = f\cdot n$
⟨*proof*⟩

lemma *wrapS-unwrapS-id*: $f\cdot\bot = \bot \Longrightarrow (wrapS\ oo\ unwrapS)\cdot f = f$
  ⟨*proof*⟩

7.3 Fibonacci example.

definition
  $\textit{fib-body} :: (Nat \to Nat) \to Nat \to Nat$ **where**
  $\textit{fib-body} \equiv \Lambda\ r.\ \textit{Nat-case}\cdot 1\cdot(\textit{Nat-case}\cdot 1\cdot(\Lambda\ n.\ r\cdot n + r\cdot(n + 1)))$
⟨*proof*⟩
definition
  $\textit{fib} :: Nat \to Nat$ **where**
  $\textit{fib} \equiv \textit{fix}\cdot\textit{fib-body}$
⟨*proof*⟩

Apply worker/wrapper.

definition
  $\textit{fib-work} :: Nat\ Stream$ **where**
  $\textit{fib-work} \equiv \textit{fix}\cdot(unwrapS\ oo\ \textit{fib-body}\ oo\ wrapS)$

definition
  $\textit{fib-wrap} :: Nat \to Nat$ **where**
  $\textit{fib-wrap} \equiv wrapS\cdot\textit{fib-work}$

lemma *wrapS-unwrapS-fib-body*: $wrapS\ oo\ unwrapS\ oo\ \textit{fib-body} = \textit{fib-body}$
⟨*proof*⟩

lemma *fib-ww-eq*: $\textit{fib} = \textit{fib-wrap}$
  ⟨*proof*⟩

Optimise.

fixrec
  $\textit{fib-work-final} :: Nat\ Stream$
and
  $\textit{fib-f-final} :: Nat \to Nat$
where
  $\textit{fib-work-final} = smap\cdot\textit{fib-f-final}\cdot nats$
$|\ \textit{fib-f-final} = \textit{Nat-case}\cdot 1\cdot(\textit{Nat-case}\cdot 1\cdot(\Lambda\ n'.\ \textit{fib-work-final}\ !!\ n' + \textit{fib-work-final}\ !!\ (n' + 1)))$

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

definition
  $\textit{fib-final} :: Nat \to Nat$ **where**
  $\textit{fib-final} \equiv \Lambda\ n.\ \textit{fib-work-final}\ !!\ n$

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*⟩

lemma *fib-final-fib-eq*: *fib-final* = *fib* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

8 Tagless interpreter via double-barreled continuations

type-synonym *'a Cont* = (*'a* → *'a*) → *'a*

definition
  val2cont :: *'a* → *'a Cont* **where**
  val2cont ≡ (Λ *a c*. *c*·*a*)

definition
  cont2val :: *'a Cont* → *'a* **where**
  cont2val ≡ (Λ *f*. *f*·*ID*)

lemma *cont2val-val2cont-id*: *cont2val oo val2cont* = *ID*
  ⟨*proof*⟩

domain *Expr* =
    Val (**lazy** *val*::*Nat*)
  | *Add* (**lazy** *addl*::*Expr*) (**lazy** *addr*::*Expr*)
  | *Throw*
  | *Catch* (**lazy** *cbody*::*Expr*) (**lazy** *chandler*::*Expr*)

fixrec *eval* :: *Expr* → *Nat Maybe*
where
  eval·(*Val*·*n*) = *Just*·*n*
| *eval*·(*Add*·*x*·*y*) = *mliftM2* (Λ *a b*. *a* + *b*)·(*eval*·*x*)·(*eval*·*y*)
| *eval*·*Throw* = *mfail*
| *eval*·(*Catch*·*x*·*y*) = *mcatch*·(*eval*·*x*)·(*eval*·*y*)

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*)

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

lemma *eval-body-strictExpr*[*simp*]: $eval\text{-}body \cdot r \cdot \bot = \bot$
⟨*proof*⟩

lemma *eval-eval-body-eq*: $eval = fix \cdot eval\text{-}body$
⟨*proof*⟩

8.1 Worker/wrapper

definition
$unwrapC :: (Expr \to Nat\ Maybe) \to (Expr \to (Nat \to Nat\ Maybe) \to Nat\ Maybe \to Nat\ Maybe)$ **where**
$unwrapC \equiv \Lambda\ g\ e\ s\ f.\ case\ g \cdot e\ of\ Nothing \Rightarrow f \mid Just \cdot n \Rightarrow s \cdot n$

lemma *unwrapC-strict*[*simp*]: $unwrapC \cdot \bot = \bot$
⟨*proof*⟩

definition
$wrapC :: (Expr \to (Nat \to Nat\ Maybe) \to Nat\ Maybe \to Nat\ Maybe) \to (Expr \to Nat\ Maybe)$ **where**
$wrapC \equiv \Lambda\ g\ e.\ g \cdot e \cdot Just \cdot Nothing$

lemma *wrapC-unwrapC-id*: $wrapC\ oo\ unwrapC = ID$
⟨*proof*⟩

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

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

fixrec $eval\text{-}body' :: (Expr \to (Nat \to Nat\ Maybe) \to Nat\ Maybe \to Nat\ Maybe)$
$\to Expr \to (Nat \to Nat\ Maybe) \to Nat\ Maybe \to Nat\ Maybe$
where
$eval\text{-}body' \cdot r \cdot (Val \cdot n) \cdot s \cdot f = s \cdot n$
$\mid eval\text{-}body' \cdot r \cdot (Add \cdot x \cdot y) \cdot s \cdot f = (case\ wrapC \cdot r \cdot x\ of$
$Nothing \Rightarrow f$
$\mid Just \cdot n \Rightarrow (case\ wrapC \cdot r \cdot y\ of$
$Nothing \Rightarrow f$
$\mid Just \cdot m \Rightarrow s \cdot (n + m)))$
$\mid eval\text{-}body' \cdot r \cdot Throw \cdot s \cdot f = f$
$\mid eval\text{-}body' \cdot r \cdot (Catch \cdot x \cdot y) \cdot s \cdot f = (case\ wrapC \cdot r \cdot x\ of$
$Nothing \Rightarrow (case\ wrapC \cdot r \cdot y\ of$
$Nothing \Rightarrow f$
$\mid Just \cdot n \Rightarrow s \cdot n)$
$\mid Just \cdot n \Rightarrow s \cdot n)$

lemma *eval-body'-strictExpr*[*simp*]: $eval\text{-}body' \cdot r \cdot \bot \cdot s \cdot f = \bot$

⟨*proof*⟩

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

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

lemma *eval-body'-eval-body-eq*: $eval\text{-}body' = unwrapC\ oo\ eval\text{-}body\ oo\ wrapC$
⟨*proof*⟩

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

lemma *eval-body-final-strictExpr*[*simp*]: $eval\text{-}body\text{-}final \cdot r \cdot \bot \cdot s \cdot f = \bot$
⟨*proof*⟩

lemma *eval-body'-eval-body-final-eq*: $eval\text{-}body\text{-}final\ oo\ unwrapC\ oo\ wrapC = eval\text{-}body'$
⟨*proof*⟩

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

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

lemma $eval = eval\text{-}final$
⟨*proof*⟩

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*⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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 ⊥. 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* → *'a llist* → *'a llist* **where**
  lappend·lnil·ys = *ys*
| *lappend·*(*lcons·x·xs*)*·ys* = *lcons·x·*(*lappend·xs·ys*)

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

fixrec *lmap* :: (*'a* → *'b*) → *'a llist* → *'b llist* **where**
  lmap·f·lnil = *lnil*
| *lmap·f·*(*lcons·x·xs*) = *lcons·*(*f·x*)*·*(*lmap·f·xs*)⟨*proof*⟩⟨*proof*⟩⟨*proof*⟩

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

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

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

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

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* → *S* → *S* **where**
  addS ≡ (Λ *x y*. *bindS·x·*(Λ *xv*. *bindS·y·*(Λ *yv*. *returnS·*(*xv* + *yv*))))

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

abbreviation $evalS :: expr\ discr \to nat\ discr\ llist$ **where**
$evalS \equiv fix \cdot evalS\text{-}body$

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 $Success = nat\ discr \to Failure \to Obs$
type-synonym $K = Success \to Failure \to Obs$

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 $unit \to Obs$.

The monad over the continuation type K is as follows:

definition $returnK :: nat\ discr \to K$ **where**
$returnK \equiv (\Lambda\ x.\ \Lambda\ s\ f.\ s \cdot x \cdot f)$

definition $bindK :: K \to (nat\ discr \to K) \to K$ **where**
$bindK \equiv \Lambda\ x\ g.\ \Lambda\ s\ f.\ x \cdot (\Lambda\ xv\ f'.\ g \cdot xv \cdot s \cdot f') \cdot f$

Our extra constants are defined as follows:

definition $addK :: K \to K \to K$ **where**

$addK \equiv (\Lambda\ x\ y.\ bindK\cdot x\cdot(\Lambda\ xv.\ bindK\cdot y\cdot(\Lambda\ yv.\ returnK\cdot(xv + yv))))$

definition $disjK :: K \to K \to K$ **where**
$disjK \equiv (\Lambda\ g\ h.\ \Lambda\ s\ f.\ g\cdot s\cdot(h\cdot s\cdot f))$

definition $failK :: K$ **where**
$failK \equiv \Lambda\ s\ f.\ f$

The continuation semantics is again straightforward:

definition
$evalK\text{-}body :: (expr\ discr \to K) \to (expr\ discr \to K)$
where
$evalK\text{-}body \equiv \Lambda\ r\ e.\ case\ undiscr\ e\ of$
$\quad const\ n \Rightarrow returnK\cdot(Discr\ n)$
$\mid add\ e1\ e2 \Rightarrow addK\cdot(r\cdot(Discr\ e1))\cdot(r\cdot(Discr\ e2))$
$\mid disj\ e1\ e2 \Rightarrow disjK\cdot(r\cdot(Discr\ e1))\cdot(r\cdot(Discr\ e2))$
$\mid fail \Rightarrow failK$

abbreviation $evalK :: expr\ discr \to K$ **where**
$evalK \equiv fix\cdot evalK\text{-}body$

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 $SK :: S \to K$ **where**
$SK\cdot lnil = failK$
$\mid SK\cdot(lcons\cdot x\cdot xs) = (\Lambda\ s\ f.\ s\cdot x\cdot(SK\cdot xs\cdot s\cdot f))$

definition
$unwrap :: (expr\ discr \to nat\ discr\ llist) \to (expr\ discr \to K)$
where
$unwrap \equiv \Lambda\ r\ e.\ SK\cdot(r\cdot e)\langle proof\rangle\langle proof\rangle$

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

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

definition $wrap :: (expr\ discr \to K) \to (expr\ discr \to nat\ discr\ llist)$ **where**
$wrap \equiv \Lambda\ r\ e.\ KS\cdot(r\cdot e)\langle proof\rangle\langle proof\rangle$

The worker/wrapper condition follows directly from these definitions.

lemma *KS-SK-id*:
$KS\cdot(SK\cdot xs) = xs$
$\langle proof\rangle$

lemma *wrap-unwrap-id*:
$wrap\ oo\ unwrap = ID$

⟨*proof*⟩

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*:
$f{\cdot}x \not\sqsubseteq g{\cdot}x \Longrightarrow f \not\sqsubseteq g$
⟨*proof*⟩

lemma *unwrap-wrap-not-under-id*:
$\mathit{unwrap}\ \mathit{oo}\ \mathit{wrap} \not\sqsubseteq \mathit{ID}$
⟨*proof*⟩

We now apply `worker_wrapper_id`:

definition *eval-work* :: *expr discr* → *K* **where**
eval-work ≡ *fix*·(*unwrap oo evalS-body oo wrap*)

definition *eval-ww* :: *expr discr* → *nat discr llist* **where**
eval-ww ≡ *wrap*·*eval-work*

lemma *evalS* = *eval-ww*
⟨*proof*⟩

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

lemma *SK-returnS-returnK*:
SK·(*returnS*·*x*) = *returnK*·*x*
⟨*proof*⟩

lemma *SK-lappend-distrib*:
SK·(*lappend*·*xs*·*ys*)·*s*·*f* = *SK*·*xs*·*s*·(*SK*·*ys*·*s*·*f*)
⟨*proof*⟩

lemma *SK-bindS-bindK*:
SK·(*bindS*·*x*·*g*) = *bindK*·(*SK*·*x*)·(*SK oo g*)
⟨*proof*⟩

lemma *SK-addS-distrib*:
SK·(*addS*·*x*·*y*) = *addK*·(*SK*·*x*)·(*SK*·*y*)
⟨*proof*⟩

lemma *SK-disjS-disjK*:
SK·(*disjS*·*xs*·*ys*) = *disjK*·(*SK*·*xs*)·(*SK*·*ys*)
⟨*proof*⟩

lemma *SK-failS-failK*:

$SK \cdot failS = failK$
⟨*proof*⟩

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

lemma *evalS-body-evalK-body*:
$unwrap\ oo\ evalS\text{-}body\ oo\ wrap = evalK\text{-}body\ oo\ unwrap\ oo\ wrap$
⟨*proof*⟩

theorem *evalS-evalK*:
$evalS = wrap \cdot evalK$
⟨*proof*⟩

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 advantange 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 $nub :: Nat\ llist \rightarrow Nat\ llist$
where
$nub \cdot lnil = lnil$
$|\ nub \cdot (x :@\ xs) = x :@\ nub \cdot (lfilter \cdot (neg\ oo\ (\Lambda\ y.\ x =_B y)) \cdot xs)$

lemma *nub-strict*[*simp*]: $nub \cdot \bot = \bot$
⟨*proof*⟩

fixrec $nub\text{-}body :: (Nat\ llist \rightarrow Nat\ llist) \rightarrow Nat\ llist \rightarrow Nat\ llist$
where
$nub\text{-}body \cdot f \cdot lnil = lnil$
$|\ nub\text{-}body \cdot f \cdot (x :@\ xs) = x :@\ f \cdot (lfilter \cdot (neg\ oo\ (\Lambda\ y.\ x =_B y)) \cdot xs)$

lemma *nub-nub-body-eq*: $nub = fix \cdot nub\text{-}body$
⟨*proof*⟩

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*·⊥ = ⊥ ⟨*proof*⟩
lemma *SetMem-SetEmpty*[*simp*]: *SetMem*·*x*·*SetEmpty* = *FF*
⟨*proof*⟩
lemma *SetMem-SetInsert*: $SetMem{\cdot}v{\cdot}(SetInsert{\cdot}x{\cdot}s) = (SetMem{\cdot}v{\cdot}s \ orelse \ x =_B v)$
⟨*proof*⟩

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*
lnil ⇒ *Nothing*
| *x* :@ *xs* ⇒ *Just*·(*x*, *R*·*xs*·(*exceptR*·*r*)))

lemma *nextR-strict1*[*simp*]: *nextR*·⊥ = ⊥ ⟨*proof*⟩
lemma *nextR-strict2*[*simp*]: *nextR*·(*R*·⊥·*S*) = ⊥ ⟨*proof*⟩

lemma *nextR-lnil*[*simp*]: *nextR*·(*R*·*lnil*·*S*) = *Nothing* ⟨*proof*⟩

definition
filterR :: *Nat* → *R* → *R* **where**
filterR ≡ (Λ *v r*. *R*·(*resultR*·*r*)·(*SetInsert*·*v*·(*exceptR*·*r*)))

definition
c2a :: *Nat llist* → *R* **where**
c2a ≡ Λ *xs*. *R*·*xs*·*SetEmpty*

definition
a2c :: *R* → *Nat llist* **where**

$a2c \equiv \Lambda\ r.\ \mathit{lfilter}\cdot(\Lambda\ v.\ \mathit{neg}\cdot(\mathit{SetMem}\cdot v\cdot(\mathit{exceptR}\cdot r)))\cdot(\mathit{resultR}\cdot r)$

lemma *a2c-strict*[*simp*]: $a2c\cdot\bot = \bot$ ⟨*proof*⟩

lemma *a2c-c2a-id*: $a2c\ oo\ c2a = ID$
⟨*proof*⟩

definition
$\mathit{wrap} :: (R \to \mathit{Nat\ llist}) \to \mathit{Nat\ llist} \to \mathit{Nat\ llist}$ **where**
$\mathit{wrap} \equiv \Lambda\ f\ xs.\ f\cdot(c2a\cdot xs)$

definition
$\mathit{unwrap} :: (\mathit{Nat\ llist} \to \mathit{Nat\ llist}) \to R \to \mathit{Nat\ llist}$ **where**
$\mathit{unwrap} \equiv \Lambda\ f\ r.\ f\cdot(a2c\cdot r)$

lemma *unwrap-strict*[*simp*]: $\mathit{unwrap}\cdot\bot = \bot$
⟨*proof*⟩

lemma *wrap-unwrap-id*: $\mathit{wrap}\ oo\ \mathit{unwrap} = ID$
⟨*proof*⟩

Equivalences needed for later.

lemma *TR-deMorgan*: $\mathit{neg}\cdot(x\ \mathit{orelse}\ y) = (\mathit{neg}\cdot x\ \mathit{andalso}\ \mathit{neg}\cdot y)$
⟨*proof*⟩

lemma *case-maybe-case*:
$(\mathit{case}\ (\mathit{case}\ L\ \mathit{of}\ \mathit{lnil} \Rightarrow \mathit{Nothing} \mid x :@\ xs \Rightarrow \mathit{Just}\cdot(h\cdot x\cdot xs))\ \mathit{of}$
$\mathit{Nothing} \Rightarrow f \mid \mathit{Just}\cdot(a,\ b) \Rightarrow g\cdot a\cdot b)$
$=$
$(\mathit{case}\ L\ \mathit{of}\ \mathit{lnil} \Rightarrow f \mid x :@\ xs \Rightarrow g\cdot(\mathit{fst}\ (h\cdot x\cdot xs))\cdot(\mathit{snd}\ (h\cdot x\cdot xs)))$
⟨*proof*⟩

lemma *case-a2c-case-caseR*:
$(\mathit{case}\ a2c\cdot w\ \mathit{of}\ \mathit{lnil} \Rightarrow f \mid x :@\ xs \Rightarrow g\cdot x\cdot xs)$
$= (\mathit{case}\ \mathit{nextR}\cdot w\ \mathit{of}\ \mathit{Nothing} \Rightarrow f \mid \mathit{Just}\cdot(x,\ r) \Rightarrow g\cdot x\cdot(a2c\cdot r))$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *filter-filterR*: $\mathit{lfilter}\cdot(\mathit{neg}\ oo\ (\Lambda\ y.\ x =_B y))\cdot(a2c\cdot r) = a2c\cdot(\mathit{filterR}\cdot x\cdot r)$
⟨*proof*⟩

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

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

lemma *nub-body-nub-body′-eq*: $\mathit{unwrap}\ oo\ \mathit{nub\text{-}body}\ oo\ \mathit{wrap} = \mathit{nub\text{-}body}'$

⟨*proof*⟩

definition
nub-body'' :: $(R \to Nat\ llist) \to R \to Nat\ llist$ **where**
nub-body'' $\equiv \Lambda\ f\ r.\ case\ nextR{\cdot}r\ of\ Nothing \Rightarrow lnil$
$\mid Just{\cdot}(x,\ xs) \Rightarrow x\ {:}@\ f{\cdot}(c2a{\cdot}(lfilter{\cdot}(neg\ oo\ (\Lambda\ y.\ x =_B\ y)){\cdot}(a2c{\cdot}xs)))$

lemma *nub-body'-nub-body''-eq*: *nub-body'* = *nub-body''*
⟨*proof*⟩

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

lemma *nub-body''-nub-body'''-eq*: *nub-body''* = *nub-body'''* *oo* (*unwrap* *oo* *wrap*)
⟨*proof*⟩

Finally glue it all together.

lemma *nub-wrap-nub-body'''*: *nub* = *wrap*$\cdot$(*fix*$\cdot$*nub-body'''*)
⟨*proof*⟩

end

11 Optimise "last".

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

11.1 The *last* function.

fixrec *llast* :: $'a\ llist \to 'a$
where
$llast{\cdot}(x\ {:}@\ yys) = (case\ yys\ of\ lnil \Rightarrow x \mid y\ {:}@\ ys \Rightarrow llast{\cdot}yys)$

lemma *llast-strict*[*simp*]: $llast{\cdot}\bot = \bot$
⟨*proof*⟩

fixrec *llast-body* :: $('a\ llist \to 'a) \to 'a\ llist \to 'a$
where
$llast\text{-}body{\cdot}f{\cdot}(x\ {:}@\ yys) = (case\ yys\ of\ lnil \Rightarrow x \mid y\ {:}@\ ys \Rightarrow f{\cdot}yys)$

lemma *llast-llast-body*: *llast* = *fix*$\cdot$*llast-body*
⟨*proof*⟩

definition *wrap* :: $('a \to 'a\ llist \to 'a) \to ('a\ llist \to 'a)$ **where**
$wrap \equiv \Lambda\ f\ (x\ {:}@\ xs).\ f{\cdot}x{\cdot}xs$

definition *unwrap* :: $('a\ llist \rightarrow 'a) \rightarrow ('a \rightarrow 'a\ llist \rightarrow 'a)$ **where**
unwrap $\equiv \Lambda\ f\ x\ xs.\ f\cdot(x\ :@\ xs)$

lemma *unwrap-strict*[*simp*]: $unwrap\cdot\bot = \bot$
⟨*proof*⟩

lemma *wrap-unwrap-ID*: *wrap oo unwrap oo llast-body* = *llast-body*
⟨*proof*⟩

definition *llast-worker* :: $('a \rightarrow 'a\ llist \rightarrow 'a) \rightarrow 'a \rightarrow 'a\ llist \rightarrow 'a$ **where**
llast-worker $\equiv \Lambda\ r\ x\ yys.$ *case yys of lnil* $\Rightarrow x \mid y\ :@\ ys \Rightarrow r\cdot y\cdot ys$

definition *llast′* :: $'a\ llist \rightarrow 'a$ **where**
llast′ $\equiv wrap\cdot(fix\cdot llast\text{-}worker)$

lemma *llast-worker-llast-body*: *llast-worker* = *unwrap oo llast-body oo wrap*
⟨*proof*⟩

lemma *llast′-llast*: *llast′* = *llast* (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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 *unwrap*s 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 $f\ \bot\ \neq\ \bot\ =\ (map\ f\ [0..])\ !!\ \bot$, where $xs\ !!\ n$ is the nth 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.