# Mechanising the worker/wrapper transformation 

Peter Gammie

September 13, 2023

## Contents

1 Introduction ..... 2
2 Fixed-point theorems for program transformation ..... 2
2.1 The rolling rule ..... 2
2.2 Least-fixed-point fusion ..... 3
3 The transformation according to Gill and Hutton ..... 4
3.1 Worker/wrapper fusion is partially correct ..... 6
3.2 A non-strict unwrap may go awry ..... 7
4 A totally-correct fusion rule ..... 9
5 Naive reverse becomes accumulator-reverse. ..... 11
5.1 Hughes lists, naive reverse, worker-wrapper optimisation. ..... 11
5.2 Gill/Hutton-style worker/wrapper. ..... 12
5.3 Optimise worker/wrapper. ..... 13
6 Unboxing types. ..... 17
6.1 Factorial example. ..... 17
6.2 Introducing an accumulator. ..... 20
7 Memoisation using streams. ..... 22
7.1 Streams ..... 22
7.2 The wrapper/unwrapper functions. ..... 23
7.3 Fibonacci example ..... 24
8 Tagless interpreter via double-barreled continuations ..... 26
8.1 Worker/wrapper ..... 27
9 Backtracking using lazy lists and continuations ..... 29
10 Transforming $O\left(n^{2}\right)$ nub into an $O(n \lg n)$ one ..... 34
10.1 The $n u b$ 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, $\Lambda_{\ldots}$ _ for continuous function abstraction, $]_{-}$for continuous function application, domain for recursive-datatype definition.

## 2 Fixed-point theorems for program transformation

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

### 2.1 The rolling rule

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

```
lemma rolling-rule-ltr: fix•(g oo \(f) \sqsubseteq g \cdot(f i x \cdot(f\) oo \(g))\)
proof -
    have \(g \cdot(f i x \cdot(f\) oo \(g)) \sqsubseteq g \cdot(f x \cdot(f\) oo \(g))\)
    by (rule below-refl) - reflexivity
    hence \(g \cdot((f\) oo \(g) \cdot(f i x \cdot(f\) oo \(g))) \sqsubseteq g \cdot(f i x \cdot(f\) oo \(g))\)
    using fix-eq[where \(F=f\) oo \(g\) ] by \(\operatorname{simp}\) - computation
    hence \((g\) oo \(f) \cdot(g \cdot(f i x \cdot(f\) oo \(g))) \sqsubseteq g \cdot(f i x \cdot(f\) oo \(g))\)
    by simp - re-associate (oo)
    thus \(f i x \cdot(g\) oo \(f) \sqsubseteq g \cdot(f i x \cdot(f\) oo \(g))\)
```

using fix-least-below by blast - induction qed
lemma rolling-rule-rtl: $g \cdot(f i x \cdot(f$ oo $g)) \sqsubseteq f i x \cdot(g$ oo $f)$
proof -
have $f x \cdot(f$ oo $g) \sqsubseteq f \cdot(f i x \cdot(g$ oo $f))$ by (rule rolling-rule-ltr $)$
hence $g \cdot(f i x \cdot(f$ oo $g)) \sqsubseteq g \cdot(f \cdot(f x \cdot(g$ oo $f)))$
by (rule monofun-cfun-arg) - g is monotonic
thus $g \cdot(f i x \cdot(f$ oo $g)) \sqsubseteq f i x \cdot(g$ oo $f)$
using fix-eq[where $F=g$ oo $f$ ] by $\operatorname{simp}$ - computation
qed
lemma rolling-rule: fix $\cdot(g$ oo $f)=g \cdot(f i x \cdot(f$ 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:

$$
\frac{a d m(\lambda x . ? P(f s t x)(\operatorname{snd} x)) \quad ? P \perp \perp \quad \bigwedge x y \cdot \frac{? P x y}{? P(? F \cdot x)(? G \cdot y)}}{? P(f i x \cdot ? F)(f i x \cdot ? G)}
$$

in a very straightforward way:

```
lemma lfp-fusion:
    assumes g.\perp = \perp
    assumes g oo f=h oo g
    shows g.(fix\cdotf) = fix\cdoth
proof(induct rule: parallel-fix-ind)
    case 2 show g
```

```
For a recursive definition comp = fix body for some body :: A->A
and a pair of functions wrap :: B->A and unwrap :: A->B where
wrap}\circ\mathrm{ unwrap = id A}\mathrm{ , we have:
    comp = wrap work
    work :: B (the worker/wrapper
    work = fix (unwrap \circ body\circ wrap)
transformation)
Also:
    (unwrap\circ wrap) work = work
    (worker/wrapper fusion)
```

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

```
case ( \(3 x y\) )
from \(\langle g \cdot x=y\rangle\langle g\) oo \(f=h\) oo \(g\rangle\) 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 oo unwrap to be the identity for all values.

```
lemma worker-wrapper-id:
    fixes wrap :: 'b::pcpo \(\rightarrow\) ' \(a::\) рсро
    fixes unwrap \(::{ }^{\prime} a \rightarrow\) ' \(b\)
    assumes wrap-unwrap: wrap oo unwrap \(=I D\)
    assumes comp-body: computation \(=f i x \cdot b o d y\)
    shows computation \(=\) wrap \(\cdot(\) fix \(\cdot(\) unwrap oo body oo wrap \())\)
proof -
    from comp-body have computation \(=f i x \cdot(I D\) oo body \()\)
        by \(\operatorname{simp}\)
```

```
    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 :: ' \(:::\) pcpo \(\rightarrow\) ' \(a::\) pcpo
    fixes unwrap :: ' \(a \rightarrow\) ' \(b\)
    assumes wrap-unwrap: wrap oo unwrap oo body \(=\) body
    assumes comp-body: computation \(=\) fix•body
    shows computation \(=\) wrap \(\cdot(\) fix \(\cdot(\) unwrap oo body oo wrap \())\)
proof -
    from comp-body have computation \(=\) fix \(\cdot(\) wrap oo unwrap oo body \()\)
        using wrap-unwrap by (simp add: assoc-oo wrap-unwrap)
    also have \(\ldots=\) wrap. \((\) fix. \(\cdot(\) 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::pсро }->\mathrm{ 'a::pсро
    fixes unwrap :: 'a }\mp@subsup{\}{}{\prime
    assumes wrap-unwrap: fix.(wrap oo unwrap oo body) = fix·body
    assumes comp-body:computation = fix·body
    shows computation = wrap.(fix.(unwrap oo body oo wrap))
proof -
    from comp-body have computation = fix.(wrap oo unwrap oo body)
        using wrap-unwrap by (simp add: assoc-oo wrap-unwrap)
    also have ... = wrap.(fix.(unwrap oo body oo wrap))
        using rolling-rule[where f=unwrap oo body and g=wrap]
        by (simp add: assoc-oo)
    finally show ?thesis.
qed
```

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

```
lemma worker-wrapper-fusion:
    fixes wrap :: 'b::pcpo \(\rightarrow\) ' \(a::\) рсро
    fixes unwrap :: ' \(a \rightarrow\) ' \(b\)
    assumes wrap-unwrap: wrap oo unwrap \(=I D\)
    assumes work: work \(=\) fix \(\cdot(\) unwrap oo body oo wrap \()\)
    shows (unwrap oo wrap)•work \(=\) work
proof -
    have (unwrap oo wrap) work \(=(\) unwrap oo wrap \() \cdot(\) fix \(\cdot(\) unwrap oo body oo wrap \())\)
        using work by simp
    also have \(\ldots=(\) unwrap oo wrap \() \cdot(\) fix \(\cdot(\) unwrap oo body oo wrap oo unwrap oo
wrap))
            using wrap-unwrap by (simp add: assoc-oo)
    also have \(\ldots=\) 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 \(\ldots=\) 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 $f=$ body yields a rule of the form 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:

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

viz that $f x=x$ implies only the partially correct fix $f \sqsubseteq x$, and not the
totally correct fix $f=x$. We use this characterisation to show that if unwrap is non-strict (i.e. 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 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 $f i x \cdot b o d y^{\prime}$, i.e. to establish:

$$
\text { fix } \cdot(\text { unwrap oo body oo wrap })=\text { fix } \cdot \text { body } y^{\prime}
$$

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: wrap oo unwrap \(=I D\)
    assumes work: work \(=\) fix \(\cdot(\) unwrap oo body oo wrap \()\)
    assumes body': unwrap oo body oo wrap \(=\) body' oo unwrap oo wrap
    shows fix•body' \(\sqsubseteq\) work
proof (rule fix-least)
    have work \(=(\) unwrap oo body oo wrap \() \cdot\) work
        using work by (simp add: fix-eq[symmetric])
    also have \(\ldots=(\) body' oo unwrap oo wrap \()\). work
        using body' by simp
    also have \(\ldots=(\) body' oo unwrap oo wrap \() \cdot((\) unwrap oo body oo wrap \() \cdot\) work \()\)
        using work by (simp add: fix-eq[symmetric])
    also have \(\ldots=(\) body' oo unwrap oo wrap oo unwrap oo body oo wrap) \(\cdot\) work
        by \(\operatorname{simp}\)
    also have \(\ldots=(\) body' oo unwrap oo body oo wrap \() \cdot\) work
        using wrap-unwrap by (simp add: assoc-oo)
    also have \(\ldots=\) body' work
        using work by (simp add: fix-eq[symmetric])
    finally show body'.work \(=\) work by \(\operatorname{simp}\)
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 $B=B(\operatorname{lazy} A)$

The functions wrap and unwrap that map between these types are routine. Note that wrap is (necessarily) strict due to the property $\forall x$. ? $f \cdot(? g \cdot x)=x$ $\Longrightarrow ? f \cdot \perp=\perp$.
fixrec wrap :: $B \rightarrow A$
where $\operatorname{wrap} \cdot(B \cdot a)=a$
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 $\cdot r=A$
The wrinkle is that the transformed worker can be strict in the recursion parameter $r$, as unwrap always lifts it.
fixrec body ${ }^{\prime}:: B \rightarrow B$
where $b_{o d y}{ }^{\prime} \cdot(B \cdot a)=B \cdot A$
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' $\sqsubseteq f i x \cdot(u n w r a p ~ o o ~ b o d y ~ o o ~ w r a p) ~$
by (rule fusion-partially-correct [OF wrap-unwrap refl body-body $\rceil$ )
However it is easy to see that it is not totally correct:

```
lemma \(\neg\) fix•(unwrap oo body oo wrap) \(\sqsubseteq f i x \cdot b o d y^{\prime}\)
proof -
    have \(l\) : fix \(\cdot(\) 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 oo unwrap $=I D$, then making unwrap strict preserves this equation:

```
lemma
    assumes wrap oo unwrap = ID
    shows wrap oo strictify.unwrap = ID
proof(rule cfun-eqI)
    fix }
    from assms
    show (wrap oo strictify\cdotunwrap)}\cdotx=ID\cdot
        by (cases }x=\perp)(\mathrm{ 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 oo unwrap $=I D$.)

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 \(\rightarrow\) ' \(a::\) рсро
    fixes unwrap :: ' \(a \rightarrow\) ' \(b\)
    fixes \(b o d y^{\prime}:: ' b \rightarrow\) ' \(b\)
    assumes wrap-unwrap: wrap oo unwrap \(=(I D:: ' a \rightarrow\) ' \(a)\)
    assumes unwrap-strict: unwrap. \(\perp=\perp\)
    assumes body-body': unwrap oo body oo wrap \(=\) body' oo (unwrap oo wrap)
    shows \(f i x \cdot b o d y=\) wrap \(\cdot(\) fix \(\cdot b o d y\) ')
proof -
    from body-body'
        have unwrap oo body oo (wrap oo unwrap) \(=\) (body' oo unwrap oo (wrap oo
unwrap))
            by (simp add: assoc-oo)
    with wrap-unwrap have unwrap oo body \(=\) body' oo unwrap
        by \(\operatorname{simp}\)
    with unwrap-strict have unwrap•(fix•body) \(=\) fix•body'
        by (rule lfp-fusion)
    hence \((\) wrap oo unwrap \() \cdot(\) fix \(\cdot\) body \()=\) wrap \(\cdot\left(\right.\) fix \(\cdot\) body \(\left.{ }^{\prime}\right)\)
        by \(\operatorname{simp}\)
    with wrap-unwrap show?thesis by simp
qed
```

We can also show a more general result which allows fusion to be optionally performed on a per-recursive-call basis using parallel_fix_ind:

```
lemma worker-wrapper-fusion-new-general:
    fixes wrap :: 'b::pcpo \(\rightarrow\) ' \(a::\) рсро
    fixes unwrap :: ' \(a \rightarrow\) ' \(b\)
    assumes wrap-unwrap: wrap oo unwrap \(=\left(I D:: ' a \rightarrow{ }^{\prime} a\right)\)
    assumes unwrap-strict: unwrap. \(\perp=\perp\)
    assumes body-body': \(\bigwedge r\). (unwrap oo wrap) \(\cdot r=r\)
                                    \(\Longrightarrow\) (unwrap oo body oo wrap) \(\cdot r=\) body' \(\cdot r\)
    shows \(f x \cdot b o d y=\) wrap \(\cdot(\) fix \(\cdot b o d y\) ')
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 (3xy)
        hence \(x y: x=y\) and unwrap-wrap: unwrap \(\cdot(\) wrap \(\cdot x)=x\) by auto
    from body-body' \(x y\) unwrap-wrap
    have (unwrap oo body oo wrap) \(\cdot x=\operatorname{body} \cdot \cdot y\)
        by simp
    moreover
    from wrap-unwrap
    have unwrap \(\cdot(\) wrap \(\cdot((\) unwrap oo body oo wrap \() \cdot x))=(\) unwrap oo body oo wrap \() \cdot x\)
        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 $=b o d y$ of type $A$ and a pair of functions wrap $:: B \rightarrow A$ and unwrap $:: A \rightarrow B$ where wrap $\circ$ unwrap $=i d_{A}$ and unwrap $\perp=\perp$, define:
comp $=$ wrap work
work $=$ unwrap (body[wrap work/comp])
(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.

```
    list2H :: 'a llist }->\mathrm{ 'a H where
    list2H \equiv lappend
lemma acc-c2a-strict[simp]:list2H\cdot\perp = \perp
    by (rule cfun-eqI, simp add: list2H-def)
definition
    H2list :: 'a H 和 a llist where
    H2list \equiv\Lambdaf.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 $\perp=(\Lambda x . \perp)$, which does not hold in Haskell.
lemma H-llist-hom-append: list2H•(xs :++ ys) = list2H•xs oo list2H•ys (is ?lhs $=$ ? $r h s$ )
proof (rule cfun-eqI)
fix $z s$
have ?lhs•zs $=(x s:++y s):++z s$ by (simp add: list2H-def)
also have $\ldots=x s:++(y s:++z s)$ by (rule lappend-assoc)
also have $\ldots=$ list2H $\cdot x s \cdot(y s:++z s)$ by (simp add: list2H-def)
also have $\ldots=$ list2H $\cdot x s \cdot($ list2H $\cdot y s \cdot z s)$ by (simp add: list2H-def)
also have $\ldots=($ list $2 H \cdot x s$ oo list2H $H$.ys) $\cdot$ zs by simp
finally show ?lhs $\cdot z s=($ list2H $\cdot x$ s oo list2H $\cdot y s) \cdot z s$.
qed
lemma H-llist-hom-id: list2H•lnil $=$ ID by (simp add: list2H-def)
lemma H2list-list2H-inv: H2list oo list2H $=I D$
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 llist $\rightarrow$ 'a llist

## where

$$
\begin{aligned}
& \operatorname{lrev} \cdot \operatorname{lnil}=\operatorname{lnil} \\
& \operatorname{lrev} \cdot(x: @ x s)=\operatorname{lrev} \cdot x s:++(x: @ \operatorname{lnil})
\end{aligned}
$$

Note "body" is the generator of lrev-def.
lemma lrev-strict[simp]: lrev $\cdot \perp=\perp$
by fixrec-simp

```
fixrec lrev-body \(::\left({ }^{\prime} a\right.\) llist \(\rightarrow\) 'a llist \() \rightarrow\) 'a llist \(\rightarrow\) 'a llist
```

where
lrev-body•r•lnil $=$ lnil
$\mid$ lrev-body $\cdot r \cdot(x: @ x s)=r \cdot x s:++(x: @ l n i l)$
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\cdotlrev-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
unwrap \(H \equiv \Lambda f x s . \operatorname{list2H} \cdot(f \cdot x s)\)
```

lemma unwrap $H$-strict [simp]: unwrap $H \cdot \perp=\perp$
unfolding unwrapH-def by (rule cfun-eqI, simp)
definition
wrap $H::\left({ }^{\prime} a\right.$ llist $\left.\rightarrow{ }^{\prime} a H\right) \rightarrow{ }^{\prime}$ 'a llist $\rightarrow$ 'a llist where
wrap $H \equiv \Lambda f x s$. H2list $\cdot(f \cdot x s)$
lemma wrapH-unwrapH-id: wrapH oo unwrapH $=I D$ (is ?lhs $=$ ? $r$ rss $)$
proof (rule cfun-eqI)+
fix $f x s$
have ?lhs $\cdot f \cdot x s=$ H2list $\cdot($ list $2 H \cdot(f \cdot x s))$ by (simp add: wrapH-def unwrapH-def)
also have $\ldots=($ H2list oo list2H $) \cdot(f \cdot x s)$ by simp
also have $\ldots=I D \cdot(f \cdot x s)$ by (simp only: H2list-list2H-inv)
also have $\ldots=$ ? $r h s \cdot f \cdot x s$ by $\operatorname{simp}$
finally show ?lhs $\cdot f \cdot x s=$ ? $r h s \cdot f \cdot x s$.
qed

### 5.2 Gill/Hutton-style worker/wrapper.

## definition

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

## definition

lrev-wrap :: 'a llist $\rightarrow$ 'a llist where
lrev-wrap $\equiv$ wrapH•lrev-work
lemma lrev-lrev-ww-eq: lrev $=$ lrev-wrap
using worker-wrapper-id[OF wrapH-unwrapH-id lrev-lrev-body-eq]
by (simp add: lrev-wrap-def lrev-work-def)

### 5.3 Optimise worker/wrapper.

Intermediate worker.
fixrec lrev-body1 :: ('a llist $\rightarrow$ ' $a H) \rightarrow$ 'a llist $\rightarrow$ 'a $H$
where
lrev-body1•r•lnil $=$ list2H $\cdot$ lnil
$\mid$ lrev-body1•r•(x:@ xs $)=$ list2H $\cdot($ wrapH $\cdot r \cdot x s:++(x: @ \operatorname{lnil}))$

## definition

lrev-work1 :: 'a llist $\rightarrow$ 'a $H$ where lrev-work1 $\equiv$ fix•lrev-body1
lemma lrev-body-lrev-body1-eq: lrev-body1 $=$ unwrapH oo lrev-body oo wrapH apply (rule cfun-eqI)+
apply (subst lrev-body.unfold)
apply (subst lrev-body1.unfold)
apply (case-tac xa)
apply (simp-all add: list2H-def wrapH-def unwrapH-def)
done
lemma lrev-work1-lrev-work-eq: lrev-work1 = lrev-work
by (unfold lrev-work-def lrev-work1-def, rule cfun-arg-cong[OF lrev-body-lrev-body1-eq])

Now use the homomorphism.
fixrec lrev-body2 $::\left({ }^{\prime} a\right.$ llist $\rightarrow$ 'a $\left.H\right) \rightarrow$ 'a llist $\rightarrow$ 'a $H$ where
lrev-body2•r•lnil $=I D$
$\mid$ lrev-body $2 \cdot r \cdot(x: @ x s)=$ list2H $\cdot($ wrapH $\cdot r \cdot x s)$ oo list2H $\cdot(x$ :@ lnil $)$
lemma lrev-body2-strict[simp]: lrev-body2•r• $\perp=\perp$
by fixrec-simp

## definition

lrev-work2 :: 'a llist $\rightarrow$ 'a $H$ where
lrev-work2 $\equiv$ fix•lrev-body2
lemma lrev-work2-strict[simp]: lrev-work2• $\perp=\perp$
unfolding lrev-work2-def
by (subst fix-eq) simp

```
lemma lrev-body2-lrev-body1-eq:lrev-body2 = lrev-body1
    by ((rule cfun-eqI)+
        , (subst lrev-body1.unfold, subst lrev-body2.unfold)
        , (simp add: H-llist-hom-append[symmetric] H-llist-hom-id))
lemma lrev-work2-lrev-work1-eq:lrev-work2 = lrev-work1
    by (unfold lrev-work2-def lrev-work1-def
        , rule cfun-arg-cong[OF lrev-body2-lrev-body1-eq])
```

Simplify.
fixrec lrev-body3 :: ('a llist $\rightarrow$ 'a $H$ ) $\rightarrow$ 'a llist $\rightarrow$ 'a $H$
where
lrev-body3•r•lnil $=I D$
|lrev-body3•r•(x:@ xs) =r•xs oo list2H•(x:@ lnil)
lemma lrev-body3-strict[simp]: lrev-body3•r• $\perp=\perp$
by fixrec-simp

## definition

```
lrev-work3 :: 'a llist \(\rightarrow\) 'a \(H\) where
lrev-work3 \(\equiv\) fix•lrev-body3
```

lemma lrev-wwfusion: list2H•((wrapH•lrev-work2) $\cdot x s)=$ lrev-work2 $\cdot x s$
proof -
\{
have list2H oo wrapH•lrev-work2 $=$ unwrapH $\cdot($ wrap $H \cdot l$ lrev-work2 $)$
by (rule cfun-eqI, simp add: unwrapH-def)
also have $\ldots=($ unwrapH oo wrapH $) \cdot$ lrev-work2 by simp
also have . . . = lrev-work2
apply -
apply (rule worker-wrapper-fusion[OF wrapH-unwrapH-id, where body=lrev-body])
apply (auto iff: lrev-body2-lrev-body1-eq lrev-body-lrev-body1-eq lrev-work2-def
lrev-work1-def)
done
finally have list2H oo wrapH•lrev-work2 = lrev-work2 .
\}
thus ?thesis using cfun-eq-iff[where $f=$ list2H oo wrapH•lrev-work2 and $g=$ lrev-work2]
by auto
qed

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

```
lemma lrev-work3 \(\sqsubseteq l r e v-w o r k 2 ~\)
    unfolding lrev-work3-def
proof(rule fix-least)
    \{
        fix \(x s\) have lrev-body3•lrev-work2•xs \(=\) lrev-work2•xs
        proof (cases xs)
            case bottom thus?thesis by simp
```

```
    next
            case lnil thus ?thesis
                unfolding lrev-work2-def
                by (subst fix-eq[where F=lrev-body2], simp)
        next
            case (lcons y ys)
            hence lrev-body3·lrev-work2.xs = lrev-work2.ys oo list2H\cdot(y :@ lnil) by simp
            also have ... = list2H·((wrapH·lrev-work2).ys) oo list2H·(y :@ lnil)
                using lrev-wwfusion[where }xs=ys] by sim
            also from lcons have ... = lrev-body2.lrev-work2.xs by simp
            also have ... = lrev-work2.xs
            unfolding lrev-work2-def by (simp only: fix-eq[symmetric])
            finally show ?thesis by simp
    qed
    }
    thus lrev-body3·lrev-work2 = lrev-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 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 \((\mathbf{i s}\) ?lhs \(=\) ?rhs)
proof (rule cfun-eqI)
    fix \(x\)
    show ?lhs \(\cdot x=\) ? \(r h s \cdot x\)
    proof (induct \(x\) )
        show lrev-work3• \(\perp=\) lrev-work2. \(\perp\)
            apply (unfold lrev-work3-def lrev-work2-def)
            apply (subst fix-eq[where \(F=\) lrev-body2])
            apply (subst fix-eq[where \(F=\) lrev-body3])
            by (simp add: lrev-body3.unfold lrev-body2.unfold)
    next
    show lrev-work3•lnil \(=\) lrev-work2.lnil
            apply (unfold lrev-work3-def lrev-work2-def)
            apply (subst fix-eq[where \(F=\) lrev-body2])
            apply (subst fix-eq[where \(F=\) lrev-body3])
            by (simp add: lrev-body3.unfold lrev-body2.unfold)
next
    fix \(a l\) assume lrev-work3 \(\cdot l=\) lrev-work2 \(\cdot l\)
    thus lrev-work3•( \(a\) :@l)=lrev-work2•(a:@l)
            apply (unfold lrev-work3-def lrev-work2-def)
            apply (subst fix-eq[where \(F=l\) lrev-body2])
            apply (subst fix-eq[where \(F=l\) lrev-body3])
```

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

Use the combined worker/wrapper-fusion rule. Note we get a weaker lemma.
lemma lrev3-2-syntactic: lrev-body3 oo (unwrapH oo wrapH) = lrev-body2
apply (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 ... $=$ wrap $H \cdot($ fix $\cdot l$ 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 $::\left({ }^{\prime} a\right.$ llist $\rightarrow$ 'a $H$ ) $\rightarrow$ 'a llist $\rightarrow$ 'a $H$
where
lrev-body-final. $\cdot \boldsymbol{r} \cdot$ lnil. $\cdot \mathrm{ys}=y s$
|lrev-body-final•r•(x:@xs)•ys=r•xs•(x:@ys)

## definition

lrev-work-final :: 'a llist $\rightarrow$ 'a $H$ where
lrev-work-final $\equiv$ fix•lrev-body-final

## definition

lrev-final :: 'a llist $\rightarrow$ 'a llist where
lrev-final $\equiv \Lambda$ xs. lrev-work-final•xs•lnil
lemma lrev-body-final-lrev-body3-eq': lrev-body-final $\cdot r \cdot x s=$ lrev-body $3 \cdot r \cdot x s$
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 lrev-final-lrev-eq: lrev \(=\) lrev-final \((\mathbf{i s}\) ?lhs \(=\) ?rhs)
proof -
    have ?lhs = lrev-wrap by (rule lrev-lrev-ww-eq)
    also have \(\ldots=\) wrapH•lrev-work by (simp only: lrev-wrap-def)
    also have \(\ldots=\) wrapH•lrev-work1 by (simp only: lrev-work1-lrev-work-eq)
    also have \(\ldots=\) wrapH•lrev-work2 by (simp only: lrev-work2-lrev-work1-eq)
    also have \(\ldots=\) wrapH•lrev-work3 by (simp only: lrev-work3-lrev-work2-eq)
    also have \(\ldots=\) wrapH•lrev-work-final by (simp only: lrev-work3-def lrev-work-final-def
lrev-body-final-lrev-body3-eq)
    also have ... = lrev-final by (simp add: lrev-final-def cfun-eqI H2list-def wrapH-def)
    finally show ?thesis.
qed
```


## 6 Unboxing types.

The original application of the worker/wrapper transformation was the unboxing of flat types by Peyton Jones and Launchbury (1991). We can model the boxed and unboxed types as (respectively) pointed and unpointed domains in HOLCF. Concretely UNat denotes the discrete domain of naturals, $U N a t_{\perp}$ the lifted (flat and pointed) variant, and Nat the standard boxed domain, isomorphic to $U N a t_{\perp}$. 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 $I D$.

The divergence monad is used to handle the unboxing, see below.

### 6.1 Factorial example.

Standard definition of factorial.

```
fixrec fac :: Nat \(\rightarrow\) 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• $\perp=\perp$
by fixrec-simp

## definition

fac-body $::($ Nat $\rightarrow$ Nat $) \rightarrow$ Nat $\rightarrow$ Nat where
fac-body $\equiv \Lambda r n$. If $n={ }_{B} 0$ then 1 else $n * r \cdot(n-1)$
lemma fac-body-strict[simp]: fac-body $\cdot r \cdot \perp=\perp$
unfolding fac-body-def by simp

```
lemma fac-fac-body-eq: fac = fix\cdotfac-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 $::($ Nat $\rightarrow$ Nat $) \rightarrow$ UNat $\rightarrow$ UNat $_{\perp}$ where
unwrap $B \equiv \Lambda f$. unbox oo $f$ oo box

```
unwrap \(B \equiv \Lambda f\). unbox oo \(f\) oo box
```

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

```
definition
    wrapB :: (UNat ->UNat })->Nat -> Nat where
    wrapB\equiv\Lambdafx.unbox}\cdotx>>=f>>=bo
```

lemma wrapB-unwrapB-body:
assumes strictF: $f \cdot \perp=\perp$
shows $($ wrapB oo unwrapB) $\cdot f=f$ (is ?lhs $=$ ? rhs $)$
proof (rule cfun-eqI)
fix $x$ :: Nat
have ?lhs $\cdot x=$ unbox $\cdot x \gg=\left(\Lambda x^{\prime}\right.$. unwrap $B \cdot f \cdot x^{\prime} \gg=$ box $)$
unfolding wrapB-def by simp
also have $\ldots=$ unbox $\cdot x \gg=\left(\Lambda x^{\prime}\right.$. unbox $\cdot\left(f \cdot\left(\right.\right.$ box $\left.\left.\cdot x^{\prime}\right)\right) \gg=$ box $)$
unfolding unwrapB-def by simp
also from strict $F$ have $\ldots=f \cdot x$ by (cases $x$, simp-all)
finally show ?lhs $\cdot x=$ ? $r h s \cdot x$.
qed

Apply worker/wrapper.

## definition

fac-work :: UNat $\rightarrow U N a t_{\perp}$ where
fac-work $\equiv$ fix•(unwrapB oo fac-body oo wrapB)

## definition

```
    fac-wrap :: Nat }->\mathrm{ Nat where
```

    fac-wrap \(\equiv\) wrap \(B \cdot f a c-w o r k\)
    ```
lemma fac-fac-ww-eq: fac \(=\) fac-wrap (is ?lhs \(=\) ? \(r h s\) )
proof -
    have wrapB oo unwrapB oo fac-body \(=\) fac-body
        using wrapB-unwrapB-body[OF fac-body-strict \(]\)
        by - (rule cfun-eqI, simp)
    thus?thesis
        using worker-wrapper-body[where computation=fac and body=fac-body and
wrap \(=\) wrap \(B\) and unwrap=unwrap \(B]\)
```

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' \(::\left(\right.\) UNat \(\rightarrow\) UNat \(\left._{\perp}\right) \rightarrow\) UNat \(\rightarrow U N a t_{\perp}\) where
fac-body \({ }^{\prime} \equiv \Lambda r n\).
        unbox. (If box \(\cdot n={ }_{B} 0\)
        then 1
        else unbox \(\cdot(\) box \(\cdot n-1) \gg=r \gg=(\Lambda\) b.box \(\cdot n *\) box \(\cdot b)\) )
```

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 $\cdot r \cdot x=$ ? rhs $\cdot r \cdot x$
using bbind-case-distr-strict $[$ where $f=\Lambda y$.box $\cdot x * y$ and $g=u n b o x \cdot(b o x \cdot x-$
1)]
bbind-case-distr-strict[where $f=\Lambda y$. box $\cdot x * y$ and $h=b o x]$
unfolding fac-body'-def fac-body-def unwrapB-def wrapB-def by simp
qed

The $u p$ constructors here again mediate the isomorphism, operationally doing nothing. Note the switch to the machine-oriented if construct: the test $n=\left(0::^{\prime} a\right)$ cannot diverge.

## definition

$$
\begin{aligned}
& \text { fac-body-final }::\left(U N a t \rightarrow U N a t_{\perp}\right) \rightarrow \text { UNat } \rightarrow \text { UNat }_{\perp} \text { where } \\
& \text { fac-body-final } \equiv \Lambda r n . \\
& \quad \text { if } n=0 \text { then up. } 1 \text { else } r \cdot(n-\# 1) \gg=\left(\Lambda \text { b. up } \cdot\left(n *_{\#} b\right)\right)
\end{aligned}
$$

lemma fac-body-final-fac-body': fac-body-final $=$ fac-body' ${ }^{\prime}(\mathbf{i s}$ ?lhs $=$ ?rhs $)$
proof(rule cfun-eqI)+
fix $r x$
show ?lhs $\cdot r \cdot x=$ ? $r h s \cdot r \cdot x$
using bbind-case-distr-strict[where $f=u n b o x$ and $g=r \cdot(x-\neq 1)$ and $h=(\Lambda b$. box $\left.\cdot\left(x *_{\#} b\right)\right)$ ]
unfolding fac-body-final-def fac-body'-def uMinus-def uMult-def zero-Nat-def one-Nat-def
by $\operatorname{simp}$
qed

## definition

fac-work-final :: UNat $\rightarrow U N a t_{\perp}$ where
fac-work-final $\equiv$ fix•fac-body-final

## definition

fac-final :: Nat $\rightarrow$ Nat where
fac-final $\equiv \Lambda$ n. unbox $\cdot n \gg=$ fac-work-final $\gg=$ box

```
lemma fac-fac-final: fac \(=\) fac-final \((\) is ?lhs=?rhs)
proof -
    have ?lhs = fac-wrap by (rule fac-fac-ww-eq)
    also have \(\ldots=\) wrap \(B \cdot f a c-\) work by (simp only: fac-wrap-def)
    also have \(\ldots=\) wrap \(B \cdot(f i x \cdot(\) unwrap \(B\) oo fac-body oo wrapB)) by (simp only:
fac-work-def)
    also have \(\ldots=\operatorname{wrap} B \cdot(f i x \cdot f a c-b o d y ')\) by (simp only: fac-body'-fac-body)
    also have \(\ldots=\) wrapB•fac-work-final by (simp only: fac-body-final-fac-body'
fac-work-final-def)
    also have \(\ldots=\) 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 \(=U N a t \rightarrow U N a t_{\perp}\)
```


## definition

```
    n2a :: UNat \(\rightarrow\) UNatAcc where
    \(n 2 a \equiv \Lambda m n . u p \cdot(m * \# n)\)
```


## definition

$a 2 n::$ UNatAcc $\rightarrow$ UNat $_{\perp}$ where
$a 2 n \equiv \Lambda a \cdot a \cdot 1$
lemma a2n-strict[simp]:a2n• $\perp=\perp$
unfolding $a 2 n$-def by simp
lemma a2n-n2a: a2n•(n2a•u) $=u p \cdot u$
unfolding a2n-def n2a-def by (simp add: uMult-arithmetic)
lemma A-hom-mult: n2a $\cdot\left(x *_{\#} y\right)=(n 2 a \cdot x>=>n 2 a \cdot y)$
unfolding n2a-def bKleisli-def by (simp add: uMult-arithmetic)
definition
unwrapA $::\left(\right.$ UNat $\rightarrow$ UNat $\left.{ }_{\perp}\right) \rightarrow$ UNat $\rightarrow$ UNatAcc where
unwrapA $\equiv \Lambda f n . f \cdot n \gg=n 2 a$
lemma unwrap $A$-strict $[\operatorname{simp}]: \operatorname{unwrap} A \cdot \perp=\perp$
unfolding unwrapA-def by (rule cfun-eqI) simp

## definition

```
    wrapA :: (UNat \(\rightarrow\) UNatAcc) \(\rightarrow\) UNat \(\rightarrow\) UNat \(\perp\) where
    wrap \(A \equiv \Lambda f\).a2n oo \(f\)
lemma wrapA-unwrapA-id: wrapA oo unwrap \(A=I D\)
    unfolding wrapA-def unwrapA-def
    apply (rule cfun-eqI)+
    apply (case-tac \(x \cdot x a\) )
    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 \(\equiv \Lambda\) rn.
        if \(n=0\) then n2a.1 else wrapA \(\cdot r \cdot(n-\# 1) \gg=\left(\Lambda\right.\) res. \(n 2 a \cdot\left(n *_{\#}\right.\) res \()\) )
```

lemma fac-acc-body1-fac-body-final-eq: fac-acc-body1 = unwrapA oo fac-body-final oo wrap $A$
unfolding fac-acc-body1-def fac-body-final-def wrapA-def unwrapA-def
by (rule cfun-eqI) + simp
Use the homomorphism.

## definition

```
fac-acc-body2 :: (UNat -> UNatAcc) -> UNat -> UNatAcc where
fac-acc-body2 \equiv \ r n.
        if }n=0\mathrm{ then n2a.1 else wrapA.r.(n-# 1) >>=(\ 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 \(\cdot n>=>r \cdot(n-\# 1)\)
    lemma fac-acc-body3-body2: fac-acc-body3 oo (unwrapA oo wrapA) $=$ fac-acc-body2 (is ? $\mathrm{lh} s=$ ? $r h s$ )
proof(rule cfun-eqI)+
fix $r n$ acc
show $((f a c-a c c-b o d y 3$ oo $($ unwrap $A$ oo wrapA $)) \cdot r \cdot n \cdot a c c)=$ fac-acc-body2 $\cdot r \cdot n \cdot a c c$
unfolding fac-acc-body2-def fac-acc-body3-def unwrapA-def
using bbind-case-distr-strict[where $f=\Lambda y . n 2 a \cdot n>=>y$ and $h=n 2 a$, sym-
metric]
by $\operatorname{simp}$
qed

```
lemma fac-work-final-body3-eq: fac-work-final = wrapA·(fix.fac-acc-body3)
    unfolding fac-work-final-def
    by (rule worker-wrapper-fusion-new[OF wrapA-unwrapA-id unwrapA-strict])
    (simp add: fac-acc-body3-body2 fac-acc-body2-body1-eq fac-acc-body1-fac-body-final-eq)
```


## definition

```
    fac-acc-body-final :: (UNat }->\mathrm{ UNatAcc) }->\mathrm{ UNat }->\mathrm{ UNatAcc where
    fac-acc-body-final \equiv\Lambda r n acc.
        if n=0 then up.acc else r}\cdot(n-#1)\cdot(n*# acc
definition
    fac-acc-work-final :: UNat }->\mathrm{ UNat }\perp\mathrm{ where
    fac-acc-work-final \equiv\Lambda x.fix.fac-acc-body-final·x·1
lemma fac-acc-work-final-fac-acc-work3-eq: fac-acc-body-final = fac-acc-body3 (is
?lhs=?rhs)
    unfolding fac-acc-body3-def fac-acc-body-final-def n2a-def bKleisli-def
    by (rule cfun-eqI)+
    (simp add: uMult-arithmetic)
lemma fac-acc-work-final-fac-work: fac-acc-work-final=fac-work-final (is ?lhs=?rhs)
proof -
    have ?rhs = wrapA·(fix·fac-acc-body3) by (rule fac-work-final-body3-eq)
    also have ... = wrapA·(fix·fac-acc-body-final)
        using fac-acc-work-final-fac-acc-work3-eq by simp
    also have ... = ?lhs
        unfolding fac-acc-work-final-def wrapA-def a2n-def
        by (simp add: cfcomp1)
    finally show ?thesis by simp
qed
```


## 7 Memoisation using streams.

### 7.1 Streams.

The type of infinite streams.
domain 'a Stream = stcons (lazy sthead :: 'a) (lazy sttail :: 'a Stream) (infixr \&\& 65)

```
fixrec smap \(::\left({ }^{\prime} a \rightarrow\right.\) ' \(\left.b\right) \rightarrow{ }^{\prime} a\) Stream \(\rightarrow\) 'b Stream
```

where
$s m a p \cdot f \cdot(x \& \& x s)=f \cdot x \& \& s m a p \cdot f \cdot x s$
lemma smap-smap: smap $\cdot f \cdot($ smap $\cdot g \cdot x s)=\operatorname{smap} \cdot(f$ oo $g) \cdot x s$
fixrec $i$-th :: ' $a$ Stream $\rightarrow$ Nat $\rightarrow{ }^{\prime} a$
where

$$
i-t h \cdot(x \& \& x s)=\text { Nat-case } \cdot x \cdot(i-t h \cdot x s)
$$

## abbreviation

$$
\begin{aligned}
& i \text {-th-syn :: 'a Stream } \Rightarrow \text { Nat } \Rightarrow{ }^{\prime} a(\text { infixl }!!100) \text { where } \\
& s!!i \equiv i \text {-th } \cdot s \cdot i
\end{aligned}
$$

The infinite stream of natural numbers.
fixrec nats :: Nat Stream
where

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

### 7.2 The wrapper/unwrapper functions.

## definition

```
unwrapS' \(::\left(\right.\) Nat \(\left.\rightarrow{ }^{\prime} a\right) \rightarrow\) 'a Stream where
unwrap \(S^{\prime} \equiv \Lambda f\).smap \(\cdot f \cdot\) nats
```

lemma unwrap $S^{\prime}$-unfold: unwrap $S^{\prime} \cdot f=f \cdot 0$ \&\& smap $\cdot(f$ oo $(\Lambda x .1+x)) \cdot$ nats fixrec unwrap $S::\left(N a t \rightarrow{ }^{\prime} a\right) \rightarrow{ }^{\prime}$ a Stream
where
unwrap $S \cdot f=f \cdot 0$ \&\& unwrap $S \cdot(f$ oo $(\Lambda x .1+x))$

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

```
lemma unwrapS-unwrap \(S^{\prime}\)-eq: unwrap \(S=\) unwrap \(S^{\prime}\left(\right.\) is ? \({ }^{2} h s=\) ?rhs \()\)
proof (rule cfun-eqI)
    fix \(f\) show ?lhs \(\cdot f=\) ? \(r h s \cdot f\)
    proof(coinduct rule: Stream.coinduct)
    let ? \(R=\lambda s s^{\prime} .(\exists f . s=f \cdot 0\) \&\& unwrapS \(\cdot(f\) oo \((\Lambda x .1+x))\)
                                    \(\wedge s^{\prime}=f \cdot 0\) \&\& smap \(\cdot(f\) oo \(\left.(\Lambda x .1+x)) \cdot n a t s\right)\)
    show Stream-bisim ?R
    proof
        fix \(s s^{\prime}\) assume ? \(R s s^{\prime}\)
        then obtain \(f\) where \(f s: s=f \cdot 0\) \&\& unwrap \(S \cdot(f\) oo \((\Lambda x .1+x))\)
                        and \(f^{\prime}: s^{\prime}=f \cdot 0 \& \& \operatorname{smap} \cdot(f\) oo \((\Lambda x .1+x)) \cdot\) nats
            by blast
        have ? \(R\) (unwrapS \(\cdot(f\) oo \((\Lambda x .1+x)))(\operatorname{smap} \cdot(f\) oo \((\Lambda x .1+x)) \cdot n a t s)\)
            by (rule exI[where \(x=f\) oo \((\Lambda x .1+x)]\)
            , subst unwrapS.unfold, subst nats.unfold, simp add: smap-smap)
        with \(f s f_{s}{ }^{\prime}\)
        show \(\left(s=\perp \wedge s^{\prime}=\perp\right)\)
            \(\vee\left(\exists h t t^{\prime}\right.\).
                    \((\exists f . t=f \cdot 0\) \&\& unwrap \(S \cdot(f\) oo \((\Lambda x .1+x))\)
                            \(\wedge t^{\prime}=f \cdot 0 \& \& \operatorname{smap} \cdot(f\) oo \(\left.(\Lambda x .1+x)) \cdot n a t s\right)\)
                            \(\left.\wedge s=h \& \& t \wedge s^{\prime}=h \& \& t^{\prime}\right)\) by best
    qed
```

```
    show ?R (?lhs·f) (?rhs·f)
    proof -
    have lhs: ?lhs.f = f.0 && unwrapS·(f oo (\Lambda x. 1 + x)) by (subst un-
wrapS.unfold, simp)
            have rhs: ?rhs\cdotf = f.0 && smap\cdot(f oo (\Lambda x. 1 + x)).nats by (rule un-
wrapS'-unfold)
            from lhs rhs show ?thesis by best
        qed
    qed
qed
definition
    wrapS :: 'a Stream }->\mathrm{ Nat }->\mp@subsup{}{}{\prime}a\mathrm{ where
    wrapS \equiv\Lambda si.s!!i
```

Note the identity requires that $f$ be strict. Gill and Hutton $(2009, \S 6.1)$ do not make this requirement, an oversight on their part.
In practice all functions worth memoising are strict in the memoised argument.
lemma wrapS-unwrapS-id':
assumes strictF: $\left(f:: N a t \rightarrow{ }^{\prime} a\right) \cdot \perp=\perp$
shows unwrapS $\cdot f!!n=f \cdot n$
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 if)
have unwrapS $\cdot f!!(i+1)=(f \cdot 0$ \&\& unwrap $S \cdot(f$ oo $(\Lambda x .1+x)))!!(i+1)$
by (subst unwrapS.unfold, simp)
also from $S u c$ have $\ldots=$ unwrapS $\cdot(f$ oo $(\Lambda x .1+x))!!i$ by simp
also from $S u c$ have $\ldots=(f$ oo $(\Lambda x .1+x)) \cdot i$ by simp
also have $\ldots=f \cdot(i+1)$ by (simp add: plus-commute)
finally show ?case .
qed
lemma wrapS-unwrapS-id: $f \cdot \perp=\perp \Longrightarrow($ wrapS oo unwrapS $) \cdot f=f$
by (rule cfun-eqI, simp add: wrapS-unwrapS-id' wrapS-def)

### 7.3 Fibonacci example.

## definition

$$
\begin{aligned}
& \text { fib-body }::(\text { Nat } \rightarrow \text { Nat }) \rightarrow \text { Nat } \rightarrow \text { Nat where } \\
& \text { fib-body } \equiv \Lambda \text { r. Nat-case } \cdot 1 \cdot(\text { Nat-case } \cdot 1 \cdot(\Lambda \text { n.r.n }+r \cdot(n+1)))
\end{aligned}
$$

## definition

fib :: Nat $\rightarrow$ Nat where
$f i b \equiv f i x \cdot f i b-b o d y$

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 $=f i b-b o d y$
proof (rule cfun-eqI)
fix $r$ show (wrapS oo unwrapS oo fib-body) $\cdot r=$ fib-body $\cdot r$ using wrapS-unwrapS-id[where $f=f i b-b o d y \cdot r]$ by simp
qed
lemma $f i b-w w-e q: f i b=f i b-w r a p$
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 $\cdot f i b-f$-final $\cdot$ nats
$\mid$ fib-f-final $=$ Nat-case $\cdot 1 \cdot\left(\right.$ Nat-case $\cdot 1 \cdot\left(\Lambda n^{\prime}\right.$. fib-work-final !! $n^{\prime}+$ fib-work-final !! $\left.\left.\left(n^{\prime}+1\right)\right)\right)$
declare fib-f-final.simps[simp del] fib-work-final.simps[simp del]

## definition

fib-final :: Nat $\rightarrow$ Nat where
fib-final $\equiv \Lambda$ 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 \((\mathbf{i s} ? l h s=? r h s)\)
proof -
    let ? \(w b=\Lambda r\). Nat-case \(\cdot 1 \cdot\left(\right.\) Nat-case \(\left.\cdot 1 \cdot\left(\Lambda n^{\prime} . r!!n^{\prime}+r!!\left(n^{\prime}+1\right)\right)\right)\)
    let ? \(m r=\Lambda(f w f::\) Nat Stream, fff). (smap•fff•nats, ? wb•fwf)
    have ?lhs = fst ( \(f i x \cdot ? m r\) )
        by (simp add: fib-work-final-def split-def csplit-def)
```

[^0]```
    also have ... = ( }\mu\mathrm{ fwf. fst (?mr.(fwf, }\mu\textrm{fff.
    using fix-cprod[where F=?mr] by simp
    also have ...=( }\mu\mathrm{ fwf. smap.( }\mu\textrm{fff.}\mathrm{ ? wb fwf).nats) by simp
    also have ... = ( }\mu\mathrm{ fwf. smap.(?wb.fwf).nats) by (simp add: fix-const)
    also have ... = ?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: fib-final = fib (is ?lhs = ?rhs)
proof -
    have ?lhs = (\Lambda n. fib-work-final !! n) by (simp add: fib-final-def)
    also have ... = ( }\Lambda\mathrm{ n. fib-work !! n) by (simp only: fib-work-final-fib-work-eq)
    also have ... = fib-wrap by (simp add: fib-wrap-def wrapS-def)
    also have ... = ?rhs by (simp only: fib-ww-eq)
    finally show ?thesis.
qed
```


## 8 Tagless interpreter via double-barreled continuations

```
type-synonym 'a Cont \(=\left({ }^{\prime} a \rightarrow{ }^{\prime} a\right) \rightarrow{ }^{\prime} a\)
```

definition

```
val2cont :: ' \(a \rightarrow\) ' \(a\) Cont where
    val2cont \(\equiv(\Lambda a c . c \cdot a)\)
```

definition
cont2val $::$ ' $a$ Cont $\rightarrow$ ' $a$ where
cont2val $\equiv(\Lambda f . f \cdot I D)$
lemma cont2val-val2cont-id: cont2val oo val2cont $=I D$
by (rule cfun-eqI, simp add: val2cont-def cont2val-def)
domain Expr $=$
Val (lazy val::Nat)
| Add (lazy addl::Expr) (lazy addr::Expr)
Throw
| Catch (lazy cbody::Expr) (lazy chandler::Expr)
fixrec eval :: Expr $\rightarrow$ Nat Maybe
where
eval $\cdot($ Val $\cdot n)=$ Just $\cdot n$
$\mid$ eval $\cdot($ Add $\cdot x \cdot y)=$ mliftM2 $(\Lambda a b . a+b) \cdot($ eval $\cdot x) \cdot($ eval $\cdot y)$
| eval. Throw $=$ mfail
$\mid$ eval $\cdot($ Catch $\cdot x \cdot y)=$ mcatch $\cdot($ eval $\cdot x) \cdot($ eval $\cdot y)$

```
fixrec eval-body \(::(\) Expr \(\rightarrow\) Nat Maybe \() \rightarrow\) Expr \(\rightarrow\) Nat Maybe
where
    eval-body \(\cdot r \cdot(\) Val \(\cdot n)=\) Just \(\cdot n\)
\(\mid\) eval-body \(\cdot r \cdot(\) Add \(\cdot x \cdot y)=\) mliftM2 \((\Lambda a b \cdot a+b) \cdot(r \cdot x) \cdot(r \cdot y)\)
| eval-body.r.Throw \(=\) mfail
\(\mid\) eval-body \(\cdot r \cdot(\) Catch \(\cdot x \cdot y)=\) mcatch \(\cdot(r \cdot x) \cdot(r \cdot y)\)
lemma eval-body-strictExpr[simp]: eval-body•r• \(\perp=\perp\)
    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

```
    unwrap \(C::(\) Expr \(\rightarrow\) Nat Maybe \() \rightarrow(\) Expr \(\rightarrow(\) Nat \(\rightarrow\) Nat Maybe \() \rightarrow\) Nat Maybe
\(\rightarrow\) Nat Maybe) where
    unwrap \(C \equiv \Lambda g\) esf.case \(g \cdot e\) of Nothing \(\Rightarrow f \mid\) Just \(\cdot n \Rightarrow s \cdot n\)
lemma unwrap \(C\)-strict \([\) simp \(]\) : unwrap \(C \cdot \perp=\perp\)
    unfolding unwrap \(C\)-def by (rule cfun-eqI) + simp
definition
    wrap \(C::(\) Expr \(\rightarrow(\) Nat \(\rightarrow\) Nat Maybe \() \rightarrow\) Nat Maybe \(\rightarrow\) Nat Maybe \() \rightarrow(\) Expr
\(\rightarrow\) Nat Maybe) where
    wrap \(C \equiv \Lambda\) ge. \(g \cdot e \cdot\) Just• \(\cdot\) Nothing
lemma wrapC-unwrapC-id: wrapC oo unwrap \(C=I D\)
proof (intro cfun-eqI)
    fix \(g e\)
    show \((\) wrap \(C\) oo unwrap \(C) \cdot g \cdot e=I D \cdot g \cdot e\)
        by (cases \(g \cdot e\), simp-all add: wrapC-def unwrapC-def)
qed
```


## definition

```
    eval-work \(::\) Expr \(\rightarrow(\) Nat \(\rightarrow\) Nat Maybe \() \rightarrow\) Nat Maybe \(\rightarrow\) Nat Maybe where
```

    eval-work \(::\) Expr \(\rightarrow(\) Nat \(\rightarrow\) Nat Maybe \() \rightarrow\) Nat Maybe \(\rightarrow\) Nat Maybe where
    eval-work \(\equiv\) fix•(unwrapC oo eval-body oo wrapC)
    ```

\section*{definition}
```

    eval-wrap \(::\) Expr \(\rightarrow\) Nat Maybe where
    eval-wrap \(\equiv\) wrapC•eval-work
    ```
fixrec eval-body' \(::(\) Expr \(\rightarrow\) (Nat \(\rightarrow\) Nat Maybe \() \rightarrow\) Nat Maybe \(\rightarrow\) Nat Maybe)
    \(\rightarrow\) Expr \(\rightarrow(\) Nat \(\rightarrow\) Nat Maybe \() \rightarrow\) Nat Maybe \(\rightarrow\) Nat Maybe
where
    eval-body' \(\cdot r \cdot(\) Val \(\cdot n) \cdot s \cdot f=s \cdot n\)
| eval-body' \(\cdot r \cdot(\) Add \(\cdot x \cdot y) \cdot s \cdot f=(\) case wrapC \(\cdot r \cdot x\) of
```

    Nothing }=>
    |ust\cdotn = (case wrapC.r.y of
Nothing }=>
|ust}\cdotm=>s\cdot(n+m))
| eval-body'r.r.Throw.s}\cdotf=
| eval-body'r.r.(Catch\cdotx\cdoty)}\cdots\cdotf=(\mathrm{ case wrapC}\cdotr\cdotx of
Nothing }=>\mathrm{ (case wrapC}C\cdotr\cdoty of
Nothing =>f
| Just.n = s.n)
| Just.n m s.n)
lemma eval-body'-strictExpr[simp]: eval-body'r.r\cdot\perp\cdots\cdotf = \perp
by (subst eval-body'.unfold, simp)

```

\section*{definition}
```

eval-work' $::$ Expr $\rightarrow($ Nat $\rightarrow$ Nat Maybe $) \rightarrow$ Nat Maybe $\rightarrow$ Nat Maybe where

```
eval-work' \(::\) Expr \(\rightarrow(\) Nat \(\rightarrow\) Nat Maybe \() \rightarrow\) Nat Maybe \(\rightarrow\) Nat Maybe where
eval-work \({ }^{\prime} \equiv\) fix•eval-body \({ }^{\prime}\)
```

eval-work ${ }^{\prime} \equiv$ fix•eval-body ${ }^{\prime}$

```

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

lemma eval-body'-eval-body-eq: eval-body' = unwrapC oo eval-body oo wrapC
apply (intro cfun-eqI)
apply (unfold unwrapC-def wrapC-def)
apply (case-tac xa)
apply simp-all
apply (simp add: wrapC-def)
apply (case-tac x.Expr1·Just.Nothing)
apply simp-all
apply (case-tac x.Expr2.Just.Nothing)
apply simp-all
apply (simp add: mfail-def)
apply (simp add: mcatch-def wrapC-def)
apply (case-tac x·Expr1·Just·Nothing)
apply simp-all
done
fixrec eval-body-final :: (Expr }->\mathrm{ (Nat }->\mathrm{ Nat Maybe) }->\mathrm{ Nat Maybe }->\mathrm{ Nat Maybe)
Axpr }->(Nat->\mathrm{ Nat Maybe ) }->\mathrm{ Nat Maybe }->\mathrm{ Nat Maybe
where
eval-body-final. }r\cdot(\mathrm{ Val }\cdotn)\cdots\cdotf=s\cdot
| eval-body-final.r}\cdot(\mathrm{ Add }\cdotx\cdoty)\cdots\cdotf=r\cdotx\cdot(\Lambda n.r\cdoty\cdot(\Lambda m.s\cdot(n+m))\cdotf)\cdot
| eval-body-final.r.Throw.s.f=f
| eval-body-final\cdotr.(Catch\cdotx\cdoty)}\cdots\cdotf=r\cdotx\cdots\cdot(r\cdoty\cdots\cdotf

```
lemma eval-body-final-strictExpr[simp]: 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: eval-body-final oo unwrapC oo wrapC \(=\) eval-body \({ }^{\prime}\)
    apply (rule cfun-eqI)+
```

    apply (case-tac xa)
        apply (simp-all add: unwrapC-def)
    done
    definition
eval-work-final :: Expr $\rightarrow$ (Nat $\rightarrow$ Nat Maybe $) \rightarrow$ Nat Maybe $\rightarrow$ Nat Maybe
where
eval-work-final $\equiv$ fix•eval-body-final
definition
eval-final :: Expr $\rightarrow$ Nat Maybe where
eval-final $\equiv(\Lambda$ 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 $\ldots=$ wrap $C \cdot($ 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 $\ldots=$ eval-final
unfolding eval-final-def eval-work-final-def wrapC-def
by $\operatorname{simp}$
finally show ?thesis.
qed

```

\section*{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 \(\mid\) add expr expr \(\mid\) disj expr expr \(\mid\) 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 $\cdot($ lcons $\cdot x \cdot x s) \cdot y s=$ lcons $\cdot x \cdot($ lappend $\cdot x s \cdot y s)$

```
fixrec lconcat :: 'a llist llist \(\rightarrow\) 'a llist where
    lconcat \(\cdot\) lnil \(=\) lnil
\(\mid\) lconcat \(\cdot(\) lcons \(\cdot x \cdot x s)=\) lappend \(\cdot x \cdot(\) lconcat \(\cdot x s)\)
fixrec lmap :: \(\left({ }^{\prime} a \rightarrow{ }^{\prime} b\right) \rightarrow\) 'a llist \(\rightarrow\) ' \(b\) llist where
    \(\operatorname{lmap} \cdot f \cdot \operatorname{lnil}=\operatorname{lnil}\)
\(\mid\) lmap \(\cdot f \cdot(\) lcons \(\cdot x \cdot x s)=\) lcons \(\cdot(f \cdot x) \cdot(\) lmap \(\cdot f \cdot x s)\)

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

type-synonym S= nat discr llist
definition returnS :: nat discr }->S\mathrm{ where
returnS = (\Lambda x.lcons }\cdotx\cdotlnil

```
definition bindS \(:: S \rightarrow(\) nat discr \(\rightarrow S) \rightarrow S\) where
    bindS \(=\left(\begin{array}{l}\text { x } \\ \mathrm{g} . \\ \text { lconcat } \cdot(\text { lmap } \cdot g \cdot x)\end{array}\right)\)

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
$a d d S \equiv(\Lambda x y \cdot \operatorname{bind} S \cdot x \cdot(\Lambda x v . \operatorname{bind} S \cdot y \cdot(\Lambda y v . \operatorname{return} S \cdot(x v+y v))))$
definition disjS :: $S \rightarrow S \rightarrow S$ where
disjS $\equiv$ lappend
definition failS :: $S$ where
failS $\equiv \operatorname{lnil}$

```

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.

\section*{definition}
\[
\left.\begin{array}{rl}
\text { evalS-body } & ::(\text { expr discr }
\end{array} \rightarrow \text { nat discr llist }\right),
\]
```

where
evalS-body \equiv\Lambdare.case undiscr e of
const n m returnS·(Discr n)
|add e1 e2 }=>\mathrm{ addS }\cdot(r\cdot(\mathrm{ Discr e1 ))}\cdot(r\cdot(\mathrm{ Discr e2 ) )
| disj e1 e2 }=>\mathrm{ disjS }\cdot(r\cdot(\mathrm{ Discr e1 ))}\cdot(r\cdot(\mathrm{ Discr e2) )
| fail = failS

```
abbreviation evalS \(::\) expr discr \(\rightarrow\) nat discr llist where
\[
e v a l S \equiv \text { fix•evalS-body }
\]

We aim to transform this evaluator into one using double-barrelled continuations; one will serve as a "success" context, taking a natural number into "the rest of the computation", and the other outright failure.
In general we could work with an arbitrary observation type ala Reynolds (1974), but for convenience we use the clearly adequate concrete type nat discr llist.
type-synonym \(O b s=\) nat discr llist
type-synonym Failure \(=O b s\)
type-synonym Success \(=\) nat discr \(\rightarrow\) Failure \(\rightarrow\) Obs
type-synonym \(K=\) Success \(\rightarrow\) Failure \(\rightarrow\) 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 \(\rightarrow\) Obs.
The monad over the continuation type \(K\) is as follows:
```

definition return $K$ :: nat discr $\rightarrow K$ where
returnK $\equiv(\Lambda x . \Lambda s f . s \cdot x \cdot f)$
definition bind $K:: K \rightarrow($ nat discr $\rightarrow K) \rightarrow K$ where
$b i n d K \equiv \Lambda x g \cdot \Lambda s f \cdot x \cdot\left(\Lambda x v f^{\prime} \cdot g \cdot x v \cdot s \cdot f^{\prime}\right) \cdot f$

```

Our extra constants are defined as follows:
```

definition $a d d K:: K \rightarrow K \rightarrow K$ where
$a d d K \equiv(\Lambda x y \cdot \operatorname{bind} K \cdot x \cdot(\Lambda x v . \operatorname{bindK} \cdot y \cdot(\Lambda y v . \operatorname{return} K \cdot(x v+y v))))$
definition disjK :: $K \rightarrow K \rightarrow K$ where
$\operatorname{disj} K \equiv(\Lambda g h . \Lambda s f \cdot g \cdot s \cdot(h \cdot s \cdot f))$
definition failK :: $K$ where
$f a i l K \equiv \Lambda s f . f$

```

The continuation semantics is again straightforward:

\section*{definition}
```

evalK-body :: (expr discr }->\mathrm{ K) }->(\mathrm{ expr discr }->\mathrm{ K)

```
```

where

```
```

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

    | fail \(\Rightarrow\) failK
    ```
abbreviation evalK :: expr discr \(\rightarrow K\) where
    evalK \(\equiv\) fix•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 $S K:: S \rightarrow K$ where
SK.lnil $=$ failK
$\mid S K \cdot($ lcons $\cdot x \cdot x s)=(\Lambda s f \cdot s \cdot x \cdot(S K \cdot x s \cdot s \cdot f))$
definition
unwrap $::($ expr discr $\rightarrow$ nat discr llist $) \rightarrow($ expr discr $\rightarrow K)$
where
unwrap $\equiv \Lambda$ re. SK $\cdot(r \cdot e)$

```

Symmetrically wrap converts an evaluator using continuations into one generating lazy lists by passing it the right continuations.
definition \(K S:: K \rightarrow S\) where
\(K S \equiv(\Lambda k . k \cdot l c o n s \cdot l n i l)\)
definition wrap :: (expr discr \(\rightarrow K) \rightarrow(\) expr discr \(\rightarrow\) nat discr llist \()\) where \(w r a p \equiv \Lambda r e . K S \cdot(r \cdot e)\)

The worker/wrapper condition follows directly from these definitions.
```

lemma $K S$-SK-id:
$K S \cdot(S K \cdot x s)=x s$
by (induct xs) (simp-all add: KS-def failK-def)

```
lemma wrap-unwrap-id:
    wrap oo unwrap \(=I D\)
    unfolding wrap-def unwrap-def
    by (simp add: KS-SK-id cfun-eq-iff)

The worker/wrapper transformation is only non-trivial if wrap and unwrap do not witness an isomorphism. In this case we can show that we do not even have a Galois connection.
lemma cfun-not-below:
\(f \cdot x \nsubseteq g \cdot x \Longrightarrow f \nsubseteq g\)
by (auto simp: cfun-below-iff)
lemma unwrap-wrap-not-under-id:
```

    unwrap oo wrap \sharpID
    proof -
let ?witness = \Lambda e.(\Lambda sf. lnil :: K)
have (unwrap oo wrap)\cdot?witness.(Discr fail)\cdot\perp\cdot(lcons\cdot0\cdotlnil)
\# ?witness·(Discr fail)\cdot\perp\cdot(lcons·O·lnil)
by (simp add: failK-def wrap-def unwrap-def KS-def)
hence (unwrap oo wrap).?witness \$ ?witness
by (fastforce intro!: cfun-not-below)
thus ?thesis by (simp add: cfun-not-below)
qed
We now apply worker_wrapper_id:
definition eval-work :: expr discr }->K\mathrm{ where
eval-work \equiv fix·(unwrap oo evalS-body oo wrap)
definition eval-ww :: expr discr }->\mathrm{ nat discr llist where
eval-ww \equiv wrap·eval-work
lemma evalS = eval-ww
unfolding eval-ww-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 \(S K\) 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:
$S K \cdot($ return $S \cdot x)=$ return $K \cdot x$
by (simp add: returnS-def returnK-def failK-def)
lemma SK-lappend-distrib:
SK $\cdot($ lappend $\cdot x s \cdot y s) \cdot s \cdot f=S K \cdot x s \cdot s \cdot(S K \cdot y s \cdot s \cdot f)$
by (induct xs) (simp-all add: failK-def)
lemma SK-bindS-bindK:
$S K \cdot(\operatorname{bindS} \cdot x \cdot g)=\operatorname{bind} K \cdot(S K \cdot x) \cdot(S K$ oo $g)$
by (induct $x$ )
(simp-all add: cfun-eq-iff
bindS-def bindK-def failK-def
SK-lappend-distrib)
lemma SK-addS-distrib:
$S K \cdot(a d d S \cdot x \cdot y)=a d d K \cdot(S K \cdot x) \cdot(S K \cdot y)$
by (clarsimp simp: cfcomp1
addS-def addK-def failK-def
SK-bindS-bindK SK-returnS-returnK)
lemma SK-disjS-disjK:

```
```

$S K \cdot(\operatorname{disjS} \cdot x s \cdot y s)=\operatorname{disjK} \cdot(S K \cdot x s) \cdot(S K \cdot y s)$
by (simp add: cfun-eq-iff disjS-def disjK-def SK-lappend-distrib)
lemma SK-failS-failK:
SK $\cdot f a i l S=$ 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 oo evalS-body oo wrap = evalK-body oo unwrap oo wrap
proof (intro cfun-eqI)
    fix \(r e^{\prime} s f\)
    obtain \(e\) :: expr
        where \(e e^{\prime}: e^{\prime}=\) Discr e by (cases \(e^{\prime}\) )
    have (unwrap oo evalS-body oo wrap) \(\cdot r \cdot(\) Discr e) \(\cdot s \cdot f\)
            \(=(\) evalK-body oo unwrap oo 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-returnS-returnK SK-addS-distrib
                                    SK-disjS-disjK SK-failS-failK)
    with \(e e^{\prime}\) show (unwrap oo evalS-body oo wrap) \(\cdot r \cdot e^{\prime} \cdot s \cdot f\)
                \(=(\) evalK-body oo unwrap oo wrap \() \cdot r \cdot e^{\prime} \cdot s \cdot f\)
        by \(\operatorname{simp}\)
qed
theorem evalS-evalK:
    evalS \(=\) wrap \(\cdot\) evalK
    using worker-wrapper-fusion-new[OF wrap-unwrap-id unwrap-strict]
        evalS-body-evalK-body
    by \(\operatorname{simp}\)

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 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.

\section*{10 Transforming \(O\left(n^{2}\right)\) nub into an \(O(n \lg n)\) one}

Andy Gill's solution, mechanised.

\subsection*{10.1 The \(n u b\) function.}
```

fixrec nub :: Nat llist $\rightarrow$ Nat llist
where
nub•lnil $=$ lnil
$\mid n u b \cdot(x: @ x s)=x: @ n u b \cdot\left(l f i l t e r \cdot\left(n e g ~ o o ~\left(\Lambda y \cdot x={ }_{B} y\right)\right) \cdot x s\right)$

```
lemma nub-strict[simp]: nub• \(\perp=\perp\)
by fixrec-simp
```

fixrec nub-body :: (Nat llist $\rightarrow$ Nat llist) $\rightarrow$ Nat llist $\rightarrow$ Nat llist

```
where
    nub-body•f•lnil \(=\operatorname{lnil}\)
\(\mid\) nub-body•f•(x :@ xs \()=x: @ f \cdot\left(l\right.\) filter \(\cdot\left(n e g\right.\) oo \(\left.\left.\left(\Lambda y . x={ }_{B} y\right)\right) \cdot x s\right)\)
lemma nub-nub-body-eq: nub \(=\) fix•nub-body
    by (rule cfun-eqI, subst nub-def, subst nub-body.unfold, simp)

\subsection*{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 \equivlnil
definition
SetInsert :: Nat }->\mathrm{ NatSet }->\mathrm{ NatSet where
SetInsert \equivlcons

```
```

definition
SetMem :: Nat }->\mathrm{ NatSet }->\mathrm{ tr where
SetMem \equivlmember.(bpred (=))
lemma SetMem-strict[simp]: SetMem·x\cdot\perp = \perp by (simp add: SetMem-def)
lemma SetMem-SetEmpty[simp]:SetMem·x\cdotSetEmpty = FF
by (simp add: SetMem-def SetEmpty-def)
lemma SetMem-SetInsert:SetMem\cdotv\cdot(SetInsert\cdotx\cdots) = (SetMem·v·s orelse x = }\mp@subsup{B}{}{\prime
v)
by (simp add: SetMem-def SetInsert-def)

```
AndyG's new type.
domain \(R=R(\mathbf{l a z y}\) result \(R::\) Nat llist) (lazy except \(R::\) NatSet)

\section*{definition}
```

next $R:: R \rightarrow(N a t * R)$ Maybe where
$n e x t R=(\Lambda r$. case ldrop While $\cdot(\Lambda$ x. SetMem $\cdot x \cdot($ except $\cdot \cdot r)) \cdot($ result $\cdot \cdot r)$ of

```
\[
\begin{aligned}
& \text { lnil } \Rightarrow \text { Nothing } \\
& \mid x: @ x s \Rightarrow \text { Just } \cdot(x, R \cdot x s \cdot(\text { exceptR•r)))}
\end{aligned}
\]
lemma nextR-strict1[simp]: nextR• \(\perp=\perp\) by (simp add: nextR-def)
lemma nextR-strict2[simp]: next \(R \cdot(R \cdot \perp \cdot S)=\perp\) by (simp add: nextR-def)
lemma next \(R\)-lnil[simp]: next \(R \cdot(R \cdot \operatorname{lnil} \cdot S)=\) Nothing by \((\operatorname{simp}\) add: next \(R-d e f)\)

\section*{definition}
```

filterR :: Nat }->R->R\mathrm{ where
filterR \equiv(\Lambda v r.R.(resultR.r)})(S\mathrm{ SetInsert v}\cdot(\mathrm{ exceptR }\cdotr))

```

\section*{definition}
\(c 2 a::\) Nat llist \(\rightarrow R\) where
\(c 2 a \equiv \Lambda x s . R \cdot x s \cdot S e t E m p t y\)

\section*{definition}
a2c \(:: R \rightarrow\) Nat llist where \(a 2 c \equiv \Lambda r\).lfilter \(\cdot(\Lambda v . n e g \cdot(\) SetMem \(\cdot v \cdot(\operatorname{exceptR} \cdot r))) \cdot(\) resultR \(\cdot r)\)
lemma a2c-strict \([\) simp \(]: a 2 c \cdot \perp=\perp\) unfolding \(a 2 c\)-def by simp
lemma a2c-c2a-id: a2c oo c2a \(=I D\)
by (rule cfun-eqI, simp add: a2c-def c2a-def lfilter-const-true)

\section*{definition}
```

wrap $::(R \rightarrow$ Nat llist $) \rightarrow$ Nat llist $\rightarrow$ Nat llist where
wrap $\equiv \Lambda f x s . f \cdot(c 2 a \cdot x s)$

```

\section*{definition}
unwrap :: (Nat llist \(\rightarrow\) Nat llist \() \rightarrow R \rightarrow\) Nat llist where unwrap \(\equiv \Lambda f r . f \cdot(a 2 c \cdot r)\)
lemma unwrap-strict[simp]: unwrap. \(\perp=\perp\) unfolding unwrap-def by (rule cfun-eqI, simp)
lemma wrap-unwrap-id: wrap oo unwrap \(=I D\)
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: neg \(\cdot(x\) orelse \(y)=(\) neg \(\cdot x\) andalso neg \(\cdot y)\)
by (rule trE [where \(p=x]\), simp-all)
lemma case-maybe-case:
```

    (case (case L of lnil \(\Rightarrow\) Nothing \(\mid x: @\) xs \(\Rightarrow\) Just \(\cdot(h \cdot x \cdot x s))\) of
        Nothing \(\Rightarrow f \mid\) Just \(\cdot(a, b) \Rightarrow g \cdot a \cdot b)\)
        \(=\)
    \((\) case \(L\) of \(\operatorname{lnil} \Rightarrow f \mid x: @ x s \Rightarrow g \cdot(f s t(h \cdot x \cdot x s)) \cdot(\operatorname{snd}(h \cdot x \cdot x s)))\)
    ```
```

    apply (cases L, simp-all)
    apply (case-tac h•a•llist)
    apply simp
    done
    lemma case-a2c-case-caseR:
(case a2c.w of lnil $\Rightarrow f \mid x: @ x s \Rightarrow g \cdot x \cdot x s)$
$=($ case nextR $\cdot w$ of Nothing $\Rightarrow f \mid$ Just $\cdot(x, r) \Rightarrow g \cdot x \cdot(a 2 c \cdot r))($ is ?lhs $=? r h s)$
proof -
have ?rhs $=($ case $($ case ldrop While $\cdot(\Lambda$ x. SetMem $\cdot x \cdot($ exceptR $\cdot w)) \cdot($ resultR $\cdot w)$ of
lnil $\Rightarrow$ Nothing
$\mid x: @ x s \Rightarrow$ Just $\cdot(x, R \cdot x s \cdot(\operatorname{exceptR} R \cdot w)))$ of Nothing $\Rightarrow f \mid$ Just $\cdot(x$,
$r) \Rightarrow g \cdot x \cdot(a 2 c \cdot r))$
by (simp add: nextR-def)
also have $\ldots=($ case ldrop While $\cdot(\Lambda$ x. SetMem $x \cdot($ except $R \cdot w)) \cdot($ resultR $\cdot w)$ of
lnil $\Rightarrow f \mid x: @ x s \Rightarrow g \cdot x \cdot(a 2 c \cdot(R \cdot x s \cdot($ except $R \cdot w))))$
using case-maybe-case $[$ where $L=l d r o p$ While $\cdot(\Lambda$ x.SetMem $\cdot x \cdot(\operatorname{exceptR} \cdot w)) \cdot($ resultR $\cdot w)$
and $f=f$ and $g=\Lambda x r . g \cdot x \cdot(a 2 c \cdot r)$ and $h=\Lambda x x s .(x$,
$R \cdot x s \cdot($ except $R \cdot w))]$
by $\operatorname{simp}$
also have $\ldots=$ ? $1 h s$
apply (simp add: a2c-def)
apply (cases resultR•w)
apply simp-all
apply (rule-tac $p=\operatorname{SetMem} \cdot a \cdot(\operatorname{except} R \cdot w)$ in $\operatorname{trE})$
apply simp-all
apply (induct-tac llist)
apply simp-all
apply (rule-tac $p=\operatorname{SetMem} \cdot a a \cdot($ exceptR $\cdot w)$ in $t r E)$
apply simp-all
done
finally show ?lhs $=$ ? $r h s$ by $\operatorname{simp}$
qed
lemma filter-filterR: lfilter $\cdot\left(\right.$ neg oo $\left.\left(\Lambda y . x={ }_{B} y\right)\right) \cdot(a 2 c \cdot r)=a 2 c \cdot(f i l t e r R \cdot x \cdot r)$
using filter-filter[where $p=\operatorname{Tr}$.neg oo $\left(\Lambda y . x={ }_{B} y\right)$ and $q=\Lambda v$. Tr.neg $\cdot($ SetMem $\cdot v \cdot($ except $R \cdot r))$ ]
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.

\section*{definition}
\[
\begin{aligned}
& \text { nub-body' }::(R \rightarrow \text { Nat llist }) \rightarrow R \rightarrow \text { Nat llist where } \\
& \text { nub-body } \equiv \Lambda \text { fr. case a2c•r of lnil } \Rightarrow \text { lnil } \\
& \qquad x: @ \text { xs } \Rightarrow x: @ f \cdot\left(\text { c2a } \cdot \left(\text { lfilter } \cdot \left(\text { neg oo } \left(\Lambda y . x={ }_{B}\right.\right.\right.\right. \\
& y)) \cdot x s))
\end{aligned}
\]
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)))\cdot(resultR.xa)
, simp-all add: fix-const)

```

\section*{definition}
```

nub-body" $::(R \rightarrow$ Nat llist $) \rightarrow R \rightarrow$ Nat llist where
nub-body ${ }^{\prime \prime} \equiv \Lambda f r$. case nextR•r of Nothing $\Rightarrow$ lnil
$\mid$ Just $\cdot(x, x s) \Rightarrow x: @ f \cdot(c 2 a \cdot($ liflter.(neg oo ( $\Lambda y . x$
$\left.\left.\left.\left.={ }_{B} y\right)\right) \cdot(a 2 c \cdot x s)\right)\right)$

```
lemma nub-body'-nub-body \({ }^{\prime \prime}\)-eq: nub-body' \(=\) nub-body"
proof (rule cfun-eqI)+
    fix \(f r\) show \(n u b-b o d y^{\prime} \cdot f \cdot r=n u b-b o d y^{\prime \prime} \cdot f \cdot r\)
        unfolding nub-body'-def nub-body' \({ }^{\prime \prime}\)-def
    using case-a2c-case-caseR[where \(f=\) lnil and \(g=\Lambda x x s . x: @ f \cdot(c 2 a \cdot(l f i l t e r \cdot(T r . n e g\)
\(\left.\left.o o\left(\Lambda y \cdot x={ }_{B} y\right)\right) \cdot x s\right)\) ) and \(w=r\) ]
    by \(\operatorname{simp}\)
qed
definition
nub-body \({ }^{\prime \prime \prime}::(R \rightarrow\) Nat llist \() \rightarrow R \rightarrow\) Nat llist where
nub-body \({ }^{\prime \prime \prime} \equiv(\Lambda f r\). case next \(R \cdot r\) of Nothing \(\Rightarrow\) lnil
\(\mid\) Just \(\cdot(x, x s) \Rightarrow x: @ f \cdot(\) filter \(R \cdot x \cdot x s))\)
lemma nub-body \({ }^{\prime \prime}\)-nub-body \({ }^{\prime \prime \prime}\)-eq: nub-body' \(=\) nub-body \({ }^{\prime \prime \prime}\) 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 ${ }^{\prime \prime \prime}$ )
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' ${ }^{\prime \prime}$-eq
nub-body'"-nub-body ${ }^{\prime \prime \prime}$-eq
by $\operatorname{simp}$

```
end

\section*{11 Optimise "last".}

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

\subsection*{11.1 The last function.}
fixrec llast :: ' \(a\) llist \(\rightarrow{ }^{\prime} a\)
where
\[
\text { llast } \cdot(x: @ y y s)=(\text { case yys of lnil } \Rightarrow x \mid y: @ y s \Rightarrow \text { llast.yys })
\]
lemma llast-strict[simp]: llast \(\cdot \perp=\perp\) by fixrec-simp
```

fixrec llast-body $::\left({ }^{\prime} a\right.$ llist $\left.\rightarrow{ }^{\prime} a\right) \rightarrow{ }^{\prime} a$ llist $\rightarrow{ }^{\prime} a$
where
llast-body $\cdot f \cdot(x: @ y y s)=($ case yys of lnil $\Rightarrow x \mid y: @ y s \Rightarrow f \cdot y y s)$
lemma llast-llast-body: llast $=$ fix•llast-body
by (rule cfun-eqI, subst llast-def, subst llast-body.unfold, simp)
definition wrap $::\left({ }^{\prime} a \rightarrow{ }^{\prime} a\right.$ llist $\left.\rightarrow{ }^{\prime} a\right) \rightarrow\left({ }^{\prime} a\right.$ llist $\left.\rightarrow{ }^{\prime} a\right)$ where
$w r a p \equiv \Lambda f(x: @ x s) . f \cdot x \cdot x s$
definition unwrap $::\left({ }^{\prime} a\right.$ llist $\left.\rightarrow{ }^{\prime} a\right) \rightarrow\left({ }^{\prime} a \rightarrow{ }^{\prime} a\right.$ llist $\rightarrow$ ' $\left.a\right)$ where
unwrap $\equiv \Lambda f x x s . f \cdot(x: @ x s)$
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 $::\left({ }^{\prime} a \rightarrow{ }^{\prime} a\right.$ llist $\rightarrow$ ' $\left.a\right) \rightarrow{ }^{\prime} a \rightarrow{ }^{\prime} a$ llist $\rightarrow$ ' $a$ where
llast-worker $\equiv \Lambda r x$ yys.case yys of lnil $\Rightarrow x \mid y: @ y s \Rightarrow r \cdot y \cdot y s$
definition llast' $::$ ' $a$ llist $\rightarrow$ ' $a$ where
llast $^{\prime} \equiv$ 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' ${ }^{\prime}$ llast: llast $^{\prime}=$ llast $($ is ?lhs $=$ ?rhs $)$
proof -
have ?rhs $=$ fix•llast-body by (simp only: llast-llast-body)
also have $\ldots$ = wrap $\cdot($ fix $\cdot($ unwrap oo llast-body oo wrap $))$
by (simp only: worker-wrapper-body[OF wrap-unwrap-ID])
also have $\ldots=$ wrap $\cdot($ fix•(llast-worker $))$
by (simp only: llast-worker-llast-body)
also have $\ldots=$ ? lhs unfolding llast'-def by simp
finally show ?thesis by simp

```

\section*{qed}
end

\section*{12 Concluding remarks}

Gill and Hutton provide two examples of fusion: accumulator introduction in their \(\S 4\), and the transformation in their \(\S 7\) 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 \(\S 5\) demonstrates the unboxing of numerical computations using a different worker/wrapper rule and does not require fusion. In their \(\S 6\) 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 \(f \perp \neq \perp=(\operatorname{map} f[0 .])!!.\perp\), where \(x s!!n\) is the \(n\)th element of \(x s\).

\section*{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 \(L N C S\). 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. McGrawHill, 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 \(L N C S\), 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 fixedpoint operators. In \(L I C S\), 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.```


[^0]:    ${ }^{1}$ The interested reader can find some historical commentary in Harel (1980); Sangiorgi (2009).

