Putting the 'K' into Bird's derivation of Knuth-Morris-Pratt string matching

Peter Gammie

March 17, 2025

Abstract

Richard Bird and collaborators have proposed a derivation of an intricate cyclic program that implements the Morris-Pratt string matching algorithm. Here we provide a proof of total correctness for Bird's derivation and complete it by adding Knuth's optimisation.

Contents

1 Introduction

We formalize a derivation of the string-matching algorithm of Knuth et al. (1977) (KMP) due to Bird (2010, Chapter 17). The central novelty of this approach is its use of a circular data structure to simultaneously compute and represent the failure function; see Figure 1 for the final program. This is challenging to model in a logic of total functions, as we discuss below, which leads us to employ the venerable machinery of domain theory.

```haskell
module KMP where

-- For testing
import Data.List ( isInfixOf )
import qualified Test.QuickCheck as QC

-- Bird's Morris-Pratt string matcher, without the 'K' optimisation
-- Chapter 17, "Pearls of Functional Algorithm Design", 2010.

data Tree a = Null
            | Node [a] (Tree a) {- ! -} (Tree a) -- remains correct with strict right subtrees

matches :: Eq a => [a] -> [a] -> [Integer]
matches ws = map fst . filter (ok . snd) . scanl step (0, root)
  where
    ok (Node vs _l _r) = null vs
    step (n, t) x = (n + 1, op t x)

    op Null _x = root
    op (Node [] l _r) x = op l x
    op (Node (v : _vs) l r) x = if x == v then r else op l x

    root = grep Null ws

    grep l [] = Node [] l Null
    grep l vvs@(v : vs) = Node vvs l (grep (op l v) vs)

-- matches [1,2,3,1,2] [1,2,1,2,3,1,2,3,1,2]

-- Our KMP (= MP with the 'K' optimisation)

kmatches :: Eq a => [a] -> [a] -> [Integer]
kmatches ws = map fst . filter (ok . snd) . scanl step (0, root)
  where
    ok (Node vs _l _r) = null vs
    step (n, t) x = (n + 1, op t x)

    op Null _x = root
    op (Node [] l _r) x = op l x
    op (Node (v : _vs) l r) x = if x == v then r else op l x

    root = grep Null ws

    next _x Null = Null
    next _x t@(Node [] _l _r) = t
    next x t@(Node (v : _vs) l _r) = if x == v then l else t

    grep l [] = Node [] l Null
    grep l vvs@(v : vs) = Node vvs (next v l) (grep (op l v) vs)

prop_matches :: [Bool] -> [Bool] -> Bool
prop_matches as bs = (as `isInfixOf` bs) == (as `matches` bs /= [])

prop_kmatches :: [Bool] -> [Bool] -> Bool
prop_kmatches as bs = (as `matches` bs) == (as `kmatches` bs)

tests :: IO ()
tests =
  do QC.quickCheck prop_matches
     QC.quickCheck prop_kmatches
```

Figure 1: Bird's KMP as a Haskell program.

Our development completes Bird's derivation of the Morris-Pratt (MP) algorithm with proofs that each derivation step preserves productivity, yielding total correctness; in other words, we show that this circular program is extensionally equal to its specification. We also add what we call the 'K' optimisation to yield the full KMP algorithm (§4.2). Our analysis inspired a Prolog implementation (§6) that some may find more perspicuous.

Here we focus on the formalities of this style of program refinement and defer further background on string matching to two excellent monographs: Gusfield (1997, §2.3) and Crochemore and Rytter (2002, §2.1). Both provide traditional presentations of the problem, the KMP algorithm and correctness proofs and complexity results. We discuss related work in §5.

1.1 Formal setting

Bird does not make his formal context explicit. The program requires non-strict datatypes and sharing to obtain the expected complexity, which implies that he is working in a lazy (call-by-need) language. For reasons we observe during our development in §4, some of Bird's definitions are difficult to make directly in Isabelle/HOL (a logic of total functions over types denoting sets) using the existing mechanisms.

We therefore adopt domain theory as mechanised by `HOLCF` (Müller et al. 1999). This logic provides a relatively straightforward if awkward way to reason about non-strict (call-by-name) programs at the cost of being too abstract to express sharing.

Bird's derivation implicitly appeals to the fold/unfold framework of Burstall and Darlington (1977), which guarantees the preservation of partial correctness: informally, if the implementation terminates then it yields a value that coincides with the specification, or implementation $\sqsubseteq$ specification in domain-theoretic terms. These rules come with side conditions that would ensure that productivity is preserved – that the implementation and specification are moreover extensionally equal – but Bird does not establish them. We note that it is easy to lose productivity through subtle uses of cyclic data structures (see §4.6 in particular), and that this derivation does not use well-known structured recursion patterns like *map* or *foldr* that mitigate these issues.

We attempt to avoid the confusions that can arise when transforming programs with named expressions (definitions or declarations) by making each step in the derivation completely self-contained: specifically, all definitions that change or depend on a definition that changes are redefined at each step. Briefly this avoids the conflation of equations with definitions; for instance, $f = f$ holds for all functions but makes for a poor definition. The issues become more subtle in the presence of recursion modelled as least fixed points, where satisfying a fixed-point equation $F f = f$ does not always imply the desired equality $f = \text{lfp } F$. Tullsen (2002) provides a fuller discussion.

As our main interest is the introduction of the circular data structure (§4.2), we choose to work with datatypes that simplify other aspects of this story. Specifically we use strict lists (§3) as they allow us to adapt many definitions and lemmas about HOL's lists and localise (the many!) definedness conditions. We also impose strong conditions on equality (§2.2) for similar reasons, and, less critically, assume products behave pleasantly (§4.1). Again Tullsen (2002) discusses how these may violate Haskell expectations.

We suggest the reader skip the next two sections and proceed to the derivation which begins in §4.

2 Extra HOLCF

lemma *lfp-fusion*:
  assumes $g\cdot\bot = \bot$
  assumes $g\ oo\ f = h\ oo\ g$
  shows $g\cdot(fix\cdot f) = fix\cdot h$
proof(*induct rule*: *parallel-fix-ind*)
  case *2* **show** $g\cdot\bot = \bot$ **by** *fact*
  case (*3 x y*)
  from ‹$g\cdot x = y$› ‹$g\ oo\ f = h\ oo\ g$› **show** $g\cdot(f\cdot x) = h\cdot y$
    by (*simp add*: *cfun-eq-iff*)
qed *simp*

lemma *predE*:
  obtains (*strict*) $p\cdot\bot = \bot$ | (*FF*) $p = (\Lambda\ x.\ FF)$ | (*TT*) $p = (\Lambda\ x.\ TT)$
using *flat-codom*[**where** f=p **and** x=$\bot$] **by** (*cases* $p\cdot\bot$; *force simp*: *cfun-eq-iff*)

lemma *retraction-cfcomp-strict*:

assumes $f\ oo\ g = ID$
shows $f{\cdot}\bot = \bot$
using *assms retraction-strict* **by** (*clarsimp simp*: *cfun-eq-iff*)

lemma *match-Pair-csplit*[*simp*]: $match\text{-}Pair{\cdot}x{\cdot}k = k{\cdot}(cfst{\cdot}x){\cdot}(csnd{\cdot}x)$
by (*cases x*) *simp*

lemmas *oo-assoc* = *assoc-oo* — Normalize name

lemma *If-cancel*[*simp*]: (*If b then x else x*) = $seq{\cdot}b{\cdot}x$
by (*cases b*) *simp-all*

lemma *seq-below*[*iff*]: $seq{\cdot}x{\cdot}y \sqsubseteq y$
by (*simp add*: *seq-conv-if*)

lemma *seq-strict-distr*: $f{\cdot}\bot = \bot \Longrightarrow seq{\cdot}x{\cdot}(f{\cdot}y) = f{\cdot}(seq{\cdot}x{\cdot}y)$
by (*cases* $x = \bot$; *clarsimp*)

lemma *strictify-below*[*iff*]: $strictify{\cdot}f \sqsubseteq f$
unfolding *strictify-def* **by** (*clarsimp simp*: *cfun-below-iff*)

lemma *If-distr*:
$[\![f\ \bot = \bot;\ cont\ f]\!] \Longrightarrow f\ (If\ b\ then\ t\ else\ e) = (If\ b\ then\ f\ t\ else\ f\ e)$
$[\![cont\ t';\ cont\ e']\!] \Longrightarrow (If\ b\ then\ t'\ else\ e')\ x = (If\ b\ then\ t'\ x\ else\ e'\ x)$
$(If\ b\ then\ t'''\ else\ e'''){\cdot}x = (If\ b\ then\ t'''{\cdot}x\ else\ e'''{\cdot}x)$
$[\![g\ \bot = \bot;\ cont\ g]\!] \Longrightarrow g\ (If\ b\ then\ t''\ else\ e'')\ y = (If\ b\ then\ g\ t''\ y\ else\ g\ e''\ y)$
by (*case-tac* [!] *b*) *simp-all*

lemma *If2-split-asm*: $P\ (If2\ Q\ x\ y) \longleftrightarrow \neg(Q = \bot \wedge \neg P\ \bot \vee Q = TT \wedge \neg P\ x \vee Q = FF \wedge \neg P\ y)$
by (*cases Q*) (*simp-all add*: *If2-def*)

lemmas *If2-splits* = *split-If2 If2-split-asm*

lemma *If2-cont*[*simp*, *cont2cont*]:
assumes *cont i*
assumes *cont t*
assumes *cont e*
shows *cont* ($\lambda x.$ *If2* (*i x*) (*t x*) (*e x*))
using *assms* **unfolding** *If2-def* **by** *simp*

lemma *If-else-FF*[*simp*]: (*If b then t else FF*) = (*b andalso t*)
by (*cases b*) *simp-all*

lemma *If-then-TT*[*simp*]: (*If b then TT else e*) = (*b orelse e*)
by (*cases b*) *simp-all*

lemma *If-cong*:
assumes $b = b'$
assumes$b = TT \Longrightarrow t = t'$
assumes $b = FF \Longrightarrow e = e'$
shows $(If\ b\ then\ t\ else\ e) = (If\ b'\ then\ t'\ else\ e')$
using *assms* **by** (*cases b*) *simp-all*

lemma *If-tr*: (*If b then t else e*) = ((*b andalso t*) *orelse* ($neg{\cdot}b$ *andalso e*))
by (*cases b*) *simp-all*

lemma *If-andalso*:
shows *If p andalso q then t else e* = *If p then If q then t else e else e*

by (*cases p*) *simp-all*

lemma *If-else-absorb*:
  assumes $c = \bot \Longrightarrow e = \bot$
  assumes $c = TT \Longrightarrow e = t$
  shows *If c then t else e* $= e$
using *assms* **by** (*cases c*; *clarsimp*)

lemma *andalso-cong*: $[\![P = P'; P' = TT \Longrightarrow Q = Q']\!] \Longrightarrow (P \textit{ andalso } Q) = (P' \textit{ andalso } Q')$
by (*cases P*) *simp-all*

lemma *andalso-weaken-left*:
  assumes $P = TT \Longrightarrow Q = TT$
  assumes $P = FF \Longrightarrow Q \neq \bot$
  assumes $P = \bot \Longrightarrow Q \neq FF$
  shows $P = (Q \textit{ andalso } P)$
using *assms* **by** (*cases P*; *cases Q*; *simp*)

lemma *orelse-cong*: $[\![P = P'; P' = FF \Longrightarrow Q = Q']\!] \Longrightarrow (P \textit{ orelse } Q) = (P' \textit{ orelse } Q')$
by (*cases P*) *simp-all*

lemma *orelse-conv*[*simp*]:
  $((x \textit{ orelse } y) = TT) \longleftrightarrow (x = TT \vee (x = FF \wedge y = TT))$
  $((x \textit{ orelse } y) = \bot) \longleftrightarrow (x = \bot \vee (x = FF \wedge y = \bot))$
by (*cases x*; *cases y*; *simp*)+

lemma *csplit-cfun2*: *cont F* $\Longrightarrow (\Lambda\ x.\ F\ x) = (\Lambda\ (x, y).\ F\ (x, y))$
by (*clarsimp simp*: *cfun-eq-iff prod-cont-iff*)

lemma *csplit-cfun3*: *cont F* $\Longrightarrow (\Lambda\ x.\ F\ x) = (\Lambda\ (x, y, z).\ F\ (x, y, z))$
by (*clarsimp simp*: *cfun-eq-iff prod-cont-iff*)

definition *convol* :: $('a::cpo \to 'b::cpo) \to ('a \to 'c::cpo) \to 'a \to 'b \times 'c$ **where**
  $convol = (\Lambda\ f\ g\ x.\ (f \cdot x,\ g \cdot x))$

abbreviation *convol-syn* :: $('a::cpo \to 'b::cpo) \Rightarrow ('a \to 'c::cpo) \Rightarrow 'a \to 'b \times 'c$ (**infix** ‹&&› *65*) **where**
  f && $g \equiv convol \cdot f \cdot g$

lemma *convol-strict*[*simp*]:
  $convol \cdot \bot \cdot \bot = \bot$
unfolding *convol-def* **by** *simp*

lemma *convol-simp*[*simp*]: $(f$ && $g) \cdot x = (f \cdot x,\ g \cdot x)$
unfolding *convol-def* **by** *simp*

definition *map-prod* :: $('a::cpo \to 'c::cpo) \to ('b::cpo \to 'd) \to 'a \times 'b \to 'c \times 'd$ **where**
  $map\text{-}prod = (\Lambda\ f\ g\ (x, y).\ (f \cdot x,\ g \cdot y))$

abbreviation *map-prod-syn* :: $('a \to 'c) \Rightarrow ('b \to 'd) \Rightarrow 'a \times 'b \to 'c \times 'd$ (**infix** ‹**› *65*) **where**
  $f ** g \equiv map\text{-}prod \cdot f \cdot g$

lemma *map-prod-cfcomp*[*simp*]: $(f ** m) \textit{ oo } (g ** n) = (f \textit{ oo } g) ** (m \textit{ oo } n)$
unfolding *map-prod-def* **by** (*clarsimp simp*: *cfun-eq-iff*)

lemma *map-prod-ID*[*simp*]: $ID ** ID = ID$
unfolding *map-prod-def* **by** (*clarsimp simp*: *cfun-eq-iff*)

lemma *map-prod-app*[*simp*]: $(f ** g) \cdot x = (f \cdot (cfst \cdot x),\ g \cdot (csnd \cdot x))$

unfolding *map-prod-def* **by** (*cases x*) (*clarsimp simp*: *cfun-eq-iff*)

lemma *map-prod-cfst*[*simp*]: *cfst oo* (*f* ∗∗ *g*) = *f oo cfst*
by (*clarsimp simp*: *cfun-eq-iff*)

lemma *map-prod-csnd*[*simp*]: *csnd oo* (*f* ∗∗ *g*) = *g oo csnd*
by (*clarsimp simp*: *cfun-eq-iff*)

2.1 Extra HOLCF Prelude.

lemma *eq-strict*[*simp*]: $eq{\cdot}(\bot{::}'a{::}Eq\text{-}strict) = \bot$
by (*simp add*: *cfun-eq-iff*)

lemma *Integer-le-both-plus-1*[*simp*]:
fixes m :: *Integer*
shows $le{\cdot}(m + 1){\cdot}(n + 1) = le{\cdot}m{\cdot}n$
by (*cases m*; *cases n*; *simp add*: *one-Integer-def*)

lemma *plus-eq-MkI-conv*:
$l + n = MkI{\cdot}m \longleftrightarrow (\exists\, l'\, n'.\ l = MkI{\cdot}l' \wedge n = MkI{\cdot}n' \wedge m = l' + n')$
by (*cases l*, *simp*) (*cases n*, *auto*)

lemma *lt-defined*:
fixes x :: *Integer*
shows
$lt{\cdot}x{\cdot}y = TT \Longrightarrow (x \neq \bot \wedge y \neq \bot)$
$lt{\cdot}x{\cdot}y = FF \Longrightarrow (x \neq \bot \wedge y \neq \bot)$
by (*cases x*; *cases y*; *clarsimp*)+

lemma *le-defined*:
fixes x :: *Integer*
shows
$le{\cdot}x{\cdot}y = TT \Longrightarrow (x \neq \bot \wedge y \neq \bot)$
$le{\cdot}x{\cdot}y = FF \Longrightarrow (x \neq \bot \wedge y \neq \bot)$
by (*cases x*; *cases y*; *clarsimp*)+

Induction on *Integer*, following the setup for the *int* type.

definition *Integer-ge-less-than* :: *int* ⇒ (*Integer* × *Integer*) *set*
where *Integer-ge-less-than d* = $\{(MkI{\cdot}z', MkI{\cdot}z)\ |z\ z'.\ d \leq z' \wedge z' < z\}$

lemma *wf-Integer-ge-less-than*: *wf* (*Integer-ge-less-than d*)
proof(*rule wf-subset*)
show *Integer-ge-less-than d* ⊆ *measure* $(\lambda z.\ nat\ (if\ z = \bot\ then\ d\ else\ (THE\ z'.\ z = MkI{\cdot}z') - d))$
unfolding *Integer-ge-less-than-def* **by** *clarsimp*
qed *simp*

2.2 Element equality

To avoid many extraneous headaches that take us far away from the interesting parts of our derivation, we assume that the elements of the pattern and text are drawn from a *pcpo* where, if the *eq* function on this type is given defined arguments, then its result is defined and coincides with (=).

Note this effectively restricts us to *flat* element types; see Paulson (1987, §4.12) for a discussion.

class *Eq-def* = *Eq-eq* +
assumes *eq-defined*: $[\![x \neq \bot; y \neq \bot]\!] \Longrightarrow eq{\cdot}x{\cdot}y \neq \bot$
begin

lemma *eq-bottom-iff*[*simp*]: $(eq{\cdot}x{\cdot}y = \bot) \longleftrightarrow (x = \bot \vee y = \bot)$
using *eq-defined* **by** *auto*

lemma *eq-defined-reflD*[*simp*]:
  $(eq{\cdot}a{\cdot}a = TT) \longleftrightarrow a \neq \bot$
  $(TT = eq{\cdot}a{\cdot}a) \longleftrightarrow a \neq \bot$
  $a \neq \bot \Longrightarrow eq{\cdot}a{\cdot}a = TT$
using *eq-refl* **by** *auto*

lemma *eq-FF*[*simp*]:
  $(FF = eq{\cdot}xs{\cdot}ys) \longleftrightarrow (xs \neq \bot \wedge ys \neq \bot \wedge xs \neq ys)$
  $(eq{\cdot}xs{\cdot}ys = FF) \longleftrightarrow (xs \neq \bot \wedge ys \neq \bot \wedge xs \neq ys)$
by (*metis* (*mono-tags*, *opaque-lifting*) *Exh-tr dist-eq-tr*(*5*) *eq-TT-dest eq-bottom-iff eq-self-neq-FF′*)+

lemma *eq-TT*[*simp*]:
  $(TT = eq{\cdot}xs{\cdot}ys) \longleftrightarrow (xs \neq \bot \wedge ys \neq \bot \wedge xs = ys)$
  $(eq{\cdot}xs{\cdot}ys = TT) \longleftrightarrow (xs \neq \bot \wedge ys \neq \bot \wedge xs = ys)$
by (*auto simp*: *local.eq-TT-dest*)

end

instance *Integer* :: *Eq-def* **by** *standard simp*

2.3 Recursive let bindings

```
Title: HOL/HOLCF/ex/Letrec.thy
Author: Brian Huffman
```

See §4.9 for an example use.

definition
  CLetrec :: (′*a*::*pcpo* → ′*a* × ′*b*::*pcpo*) → ′*b* **where**
  CLetrec = (Λ *F*. *prod.snd* (*F*·(μ *x*. *prod.fst* (*F*·*x*))))

nonterminal *recbinds* **and** *recbindt* **and** *recbind*

syntax
  -recbind :: *logic* ⇒ *logic* ⇒ *recbind* (‹(‹*indent=2 notation=*‹*mixfix Letrec binding*››- =/ -)› *10*)
  :: *recbind* ⇒ *recbindt* (‹-›)
  -recbindt :: *recbind* ⇒ *recbindt* ⇒ *recbindt* (‹-,/ -›)
  :: *recbindt* ⇒ *recbinds* (‹-›)
  -recbinds :: *recbindt* ⇒ *recbinds* ⇒ *recbinds* (‹-;/ -›)
  -Letrec :: *recbinds* ⇒ *logic* ⇒ *logic* (‹(‹*notation=*‹*mixfix Letrec expression*››*Letrec* (-)/ *in* (-))› *10*)

syntax-consts
  -recbind -recbindt -recbinds -Letrec == *CLetrec*

translations
  (*recbindt*) *x* = *a*, (*y*,*ys*) = (*b*,*bs*) == (*recbindt*) (*x*,*y*,*ys*) = (*a*,*b*,*bs*)
  (*recbindt*) *x* = *a*, *y* = *b* == (*recbindt*) (*x*,*y*) = (*a*,*b*)

translations
  -Letrec (*-recbinds b bs*) *e* == *-Letrec b* (*-Letrec bs e*)
  Letrec xs = *a in* (*e*,*es*) == *CONST CLetrec*·(Λ *xs*. (*a*,*e*,*es*))
  Letrec xs = *a in e* == *CONST CLetrec*·(Λ *xs*. (*a*,*e*))

3 Strict lists

Head- and tail-strict lists. Many technical Isabelle details are lifted from *HOLCF−Prelude.Data-List*; names follow HOL, prefixed with *s*.

domain *'a slist* (‹[:-:]›) =
  snil (‹[::]›)
| *scons* (*shead* :: *'a*) (*stail* :: *'a slist*) (**infixr** ‹:#› *65*)

lemma *scons-strict*[*simp*]: *scons*·⊥ = ⊥
by (*clarsimp simp*: *cfun-eq-iff*)

lemma *shead-bottom-iff*[*simp*]: (*shead*·*xs* = ⊥) ⟷ (*xs* = ⊥ ∨ *xs* = [::])
by (*cases xs*) *simp-all*

lemma *stail-bottom-iff*[*simp*]: (*stail*·*xs* = ⊥) ⟷ (*xs* = ⊥ ∨ *xs* = [::])
by (*cases xs*) *simp-all*

lemma *match-snil-match-scons-slist-case*: *match-snil*·*xs*·*k1* +++ *match-scons*·*xs*·*k2* = *slist-case*·*k1*·*k2*·*xs*
by (*cases xs*) *simp-all*

lemma *slist-bottom'*: *slist-case*·⊥·⊥·*xs* = ⊥
by (*cases xs*; *simp*)

lemma *slist-bottom*[*simp*]: *slist-case*·⊥·⊥ = ⊥
by (*simp add*: *cfun-eq-iff slist-bottom'*)

lemma *slist-case-distr*:
  f·⊥ = ⊥ ⟹ *f*·(*slist-case*·*g*·*h*·*xs*) = *slist-case*·(*f*·*g*)·(Λ *x xs*. *f*·(*h*·*x*·*xs*))·*xs*
  slist-case·*g'*·*h'*·*xs*·*z* = *slist-case*·(*g'*·*z*)·(Λ *x xs*. *h'*·*x*·*xs*·*z*)·*xs*
by (*case-tac* [!] *xs*) *simp-all*

lemma *slist-case-cong*:
  assumes *xs* = *xs'*
  assumes *xs'* = [::] ⟹ *n* = *n'*
  assumes ⋀*y ys*. ⟦*xs'* = *y* :# *ys*; *y* ≠ ⊥; *ys* ≠ ⊥⟧ ⟹ *c y ys* = *c' y ys*
  assumes *cont* (λ(*x*, *y*). *c x y*)
  assumes *cont* (λ(*x*, *y*). *c' x y*)
  shows *slist-case*·*n*·(Λ *x xs*. *c x xs*)·*xs* = *slist-case*·*n'*·(Λ *x xs*. *c' x xs*)·*xs'*
using *assms* **by** (*cases xs*; *cases xs'*; *clarsimp simp*: *prod-cont-iff*)

Section syntax for *scons* ala Haskell.

syntax
  -scons-section :: *'a* → [:*'a*:] → [:*'a*:] (‹'(:#')›)
  -scons-section-left :: *'a* ⇒ [:*'a*:] → [:*'a*:] (‹'(-:#')›)
syntax-consts
  -scons-section-left == *scons*
translations
  (*x*:#) == (*CONST Rep-cfun*) (*CONST scons*) *x*

abbreviation *scons-section-right* :: [:*'a*:] ⇒ *'a* → [:*'a*:] (‹'(:#-')›) **where**
  (:#*xs*) ≡ Λ *x*. *x* :# *xs*

syntax
  -strict-list :: *args* ⇒ [:*'a*:] (‹[:(-):]›)
syntax-consts
  -strict-list == *scons*
translations
  [:*x*, *xs*:] == *x* :# [:*xs*:]

```
  [:x:] == x :# [::]
```

Class instances.

```
instantiation slist :: (Eq) Eq-strict
begin

fixrec eq-slist :: [:'a:] → [:'a:] → tr where
  eq-slist·[::]·[::] = TT
| ⟦x ≠ ⊥; xs ≠ ⊥⟧ ⟹ eq-slist·(x :# xs)·[::] = FF
| ⟦y ≠ ⊥; ys ≠ ⊥⟧ ⟹ eq-slist·[::]·(y :# ys) = FF
| ⟦x ≠ ⊥; xs ≠ ⊥; y ≠ ⊥; ys ≠ ⊥⟧ ⟹ eq-slist·(x :# xs)·(y :# ys) = (eq·x·y andalso eq-slist·xs·ys)

instance proof
  fix xs :: [:'a:]
  show eq·xs·⊥ = ⊥
    by (cases xs) (subst eq-slist.unfold; simp)+
  show eq·⊥·xs = ⊥
    by (cases xs) (subst eq-slist.unfold; simp)+
qed

end

instance slist :: (Eq-sym) Eq-sym
proof
  fix xs ys :: [:'a:]
  show eq·xs·ys = eq·ys·xs
  proof (induct xs arbitrary: ys)
    case snil
    show ?case by (cases ys; simp)
  next
    case scons
    then show ?case by (cases ys; simp add: eq-sym)
  qed simp-all
qed

instance slist :: (Eq-equiv) Eq-equiv
proof
  fix xs ys zs :: [:'a:]
  show eq·xs·xs ≠ FF
    by (induct xs) simp-all
  assume eq·xs·ys = TT and eq·ys·zs = TT then show eq·xs·zs = TT
  proof (induct xs arbitrary: ys zs)
    case (snil ys zs) then show ?case by (cases ys, simp-all)
  next
    case (scons x xs ys zs) with eq-trans show ?case
      by (cases ys; cases zs) auto
  qed simp-all
qed

instance slist :: (Eq-eq) Eq-eq
proof
  fix xs ys :: [:'a:]
  show eq·xs·xs ≠ FF
    by (induct xs) simp-all
  assume eq·xs·ys = TT then show xs = ys
  proof (induct xs arbitrary: ys)
    case (snil ys) then show ?case by (cases ys) simp-all
  next
```

case (*scons x xs ys*) **then show** *?case* **by** (*cases ys*) *auto*
qed *simp*
qed

instance *slist* :: (*Eq-def*) *Eq-def*
proof
fix *xs ys* :: [:'a:]
assume $xs \neq \bot$ **and** $ys \neq \bot$
then show $eq \cdot xs \cdot ys \neq \bot$
proof(*induct xs arbitrary*: *ys*)
case (*snil ys*) **then show** *?case* **by** (*cases ys*) *simp-all*
next
case (*scons a xs*) **then show** *?case* **by** (*cases ys*) *simp-all*
qed *simp*
qed

lemma *slist-eq-TT-snil*[*simp*]:
fixes *xs* :: [:'a::Eq:]
shows $(eq \cdot xs \cdot [::] = TT) \longleftrightarrow (xs = [::])$
$(eq \cdot [::] \cdot xs = TT) \longleftrightarrow (xs = [::])$
by (*cases xs*; *simp*)+

lemma *slist-eq-FF-snil*[*simp*]:
fixes *xs* :: [:'a::Eq:]
shows $(eq \cdot xs \cdot [::] = FF) \longleftrightarrow (\exists\, y\ ys.\ y \neq \bot \wedge ys \neq \bot \wedge xs = y :\#\ ys)$
$(eq \cdot [::] \cdot xs = FF) \longleftrightarrow (\exists\, y\ ys.\ y \neq \bot \wedge ys \neq \bot \wedge xs = y :\#\ ys)$
by (*cases xs*; *force*)+

3.1 Some of the usual reasoning infrastructure

inductive *slistmem* :: $'a \Rightarrow [:'a:] \Rightarrow bool$ **where**
$[\![x \neq \bot;\ xs \neq \bot]\!] \Longrightarrow slistmem\ x\ (x :\#\ xs)$
$|\ [\![slistmem\ x\ xs;\ y \neq \bot]\!] \Longrightarrow slistmem\ x\ (y :\#\ xs)$

lemma *slistmem-bottom1*[*iff*]:
fixes *x* :: 'a
shows $\neg\ slistmem\ x\ \bot$
by *rule* (*induct x* ⊥::[:'a:] *rule*: *slistmem.induct*; *fastforce*)

lemma *slistmem-bottom2*[*iff*]:
fixes *xs* :: [:'a:]
shows $\neg\ slistmem\ \bot\ xs$
by *rule* (*induct* ⊥::'a *xs rule*: *slistmem.induct*; *fastforce*)

lemma *slistmem-nil*[*iff*]:
shows $\neg\ slistmem\ x\ [::]$
by (*fastforce elim*: *slistmem.cases*)

lemma *slistmem-scons*[*simp*]:
shows $slistmem\ x\ (y :\#\ ys) \longleftrightarrow (x = y \wedge x \neq \bot \wedge ys \neq \bot) \vee (slistmem\ x\ ys \wedge y \neq \bot)$
proof −
have $x = y \vee slistmem\ x\ ys$ **if** $slistmem\ x\ (y :\#\ ys)$
using *that* **by** (*induct x y* :# *ys arbitrary*: *y ys rule*: *slistmem.induct*; *force*)
then show *?thesis* **by** (*auto elim*: *slistmem.cases intro*: *slistmem.intros*)
qed

definition *sset* :: $[:'a:] \Rightarrow 'a\ set$ **where**
$sset\ xs = \{x.\ slistmem\ x\ xs\}$

lemma *sset-simp*[*simp*]:
  shows *sset* $\bot = \{\}$
    and *sset* $[::] = \{\}$
    and $[\![x \neq \bot; xs \neq \bot]\!] \Longrightarrow sset\ (x :\#\ xs) = insert\ x\ (sset\ xs)$
unfolding *sset-def* **by** (*auto elim*: *slistmem.cases intro*: *slistmem.intros*)

lemma *sset-defined*[*simp*]:
  assumes $x \in sset\ xs$
  shows $x \neq \bot$
using *assms sset-def* **by** *force*

lemma *sset-below*:
  assumes $y \in sset\ ys$
  assumes $xs \sqsubseteq ys$
  assumes $xs \neq \bot$
  obtains x **where** $x \in sset\ xs$ **and** $x \sqsubseteq y$
using *assms*
proof(*induct ys arbitrary*: *xs*)
  case (*scons y ys xs*) **then show** *?case* **by** (*cases xs*) *auto*
qed *simp-all*

3.2 Some of the usual operations

A variety of functions on lists. Drawn from Bird (1987), *HOL.List* and *HOLCF−Prelude.Data-List*. The definitions vary because, for instance, the strictness of some of those in *HOLCF−Prelude.Data-List* correspond neither to those in Haskell nor Bird's expectations (specifically *stails*, *inits*, *sscanl*).

fixrec *snull* :: $[:'a:] \to tr$ **where**
  $snull{\cdot}[::] = TT$
| $[\![x \neq \bot; xs \neq \bot]\!] \Longrightarrow snull{\cdot}(x :\#\ xs) = FF$

lemma *snull-strict*[*simp*]: $snull{\cdot}\bot = \bot$
by *fixrec-simp*

lemma *snull-bottom-iff*[*simp*]: $(snull{\cdot}xs = \bot) \longleftrightarrow (xs = \bot)$
by (*cases xs*) *simp-all*

lemma *snull-FF-conv*: $(snull{\cdot}xxs = FF) \longleftrightarrow (\exists\, x\ xs.\ xxs \neq \bot \land xxs = x :\#\ xs)$
by (*cases xxs*) *simp-all*

lemma *snull-TT-conv*[*simp*]: $(snull{\cdot}xs = TT) \longleftrightarrow (xs = [::])$
by (*cases xs*) *simp-all*

lemma *snull-eq-snil*: $snull{\cdot}xs = eq{\cdot}xs{\cdot}[::]$
by (*cases xs*) *simp-all*

fixrec *smap* :: $('a \to 'b) \to [:'a:] \to [:'b:]$ **where**
  $smap{\cdot}f{\cdot}[::] = [::]$
| $[\![x \neq\bot; xs \neq \bot]\!] \Longrightarrow smap{\cdot}f{\cdot}(x :\#\ xs) = f{\cdot}x :\#\ smap{\cdot}f{\cdot}xs$

lemma *smap-strict*[*simp*]: $smap{\cdot}f{\cdot}\bot = \bot$
by *fixrec-simp*

lemma *smap-bottom-iff*[*simp*]: $(smap{\cdot}f{\cdot}xs = \bot) \longleftrightarrow (xs = \bot \lor (\exists\, x{\in}sset\ xs.\ f{\cdot}x = \bot))$
by (*induct xs*) *simp-all*

lemma *smap-is-snil-conv*[*simp*]:
  $(smap{\cdot}f{\cdot}xs = [::]) \longleftrightarrow (xs = [::])$

```
  ( [::] = smap·f·xs) ⟷ (xs = [::])
by (cases xs; simp)+

lemma smap-strict-scons[simp]:
  assumes f·⊥ = ⊥
  shows smap·f·(x :# xs) = f·x :# smap·f·xs
using assms by (cases x :# xs = ⊥; fastforce)

lemma smap-ID': smap·ID·xs = xs
by (induct xs) simp-all

lemma smap-ID[simp]: smap·ID = ID
by (clarsimp simp: cfun-eq-iff smap-ID')

lemma smap-cong:
  assumes xs = xs'
  assumes ⋀x. x ∈ sset xs ⟹ f·x = f'·x
  shows smap·f·xs = smap·f'·xs'
using assms by (induct xs arbitrary: xs') auto

lemma smap-smap'[simp]:
  assumes f·⊥ = ⊥
  shows smap·f·(smap·g·xs) = smap·(f oo g)·xs
using assms by (induct xs) simp-all

lemma smap-smap[simp]:
  assumes f·⊥ = ⊥
  shows smap·f oo smap·g = smap·(f oo g)
using assms by (clarsimp simp: cfun-eq-iff)

lemma sset-smap[simp]:
  assumes ⋀x. x ∈ sset xs ⟹ f·x ≠ ⊥
  shows sset (smap·f·xs) = { f·x | x. x ∈ sset xs }
using assms by (induct xs) auto

lemma shead-smap-distr:
  assumes f·⊥ = ⊥
  assumes ⋀x. x∈sset xs ⟹ f·x ≠ ⊥
  shows shead·(smap·f·xs) = f·(shead·xs)
using assms by (induct xs) simp-all

fixrec sappend :: [:'a:] → [:'a:] → [:'a:] where
  sappend·[::]·ys = ys
| ⟦x ≠ ⊥; xs ≠ ⊥⟧ ⟹ sappend·(x :# xs)·ys = x :# sappend·xs·ys

abbreviation sappend-syn :: 'a slist ⇒ 'a slist ⇒ 'a slist (infixr ‹:@› 65) where
  xs :@ ys ≡ sappend·xs·ys

lemma sappend-strict[simp]: sappend·⊥ = ⊥
by fixrec-simp

lemma sappend-strict2[simp]: xs :@ ⊥ = ⊥
by (induct xs) simp-all

lemma sappend-bottom-iff[simp]: (xs :@ ys = ⊥) ⟷ (xs = ⊥ ∨ ys = ⊥)
by (induct xs) simp-all

lemma sappend-scons[simp]: (x :# xs) :@ ys = x :# xs :@ ys
```

by (*cases* x :# xs = ⊥; *fastforce*)

lemma *sappend-assoc*[*simp*]: (xs :@ ys) :@ zs = xs :@ (ys :@ zs)
by (*induct xs*) *simp-all*

lemma *sappend-snil-id-left*[*simp*]: *sappend*·[::] = *ID*
by (*simp add*: *cfun-eq-iff*)

lemma *sappend-snil-id-right*[*iff*]: xs :@ [::] = xs
by (*induct xs*) *simp-all*

lemma *snil-append-iff*[*iff*]: xs :@ ys = [::] ⟷ xs = [::] ∧ ys = [::]
by (*induct xs*) *simp-all*

lemma *smap-sappend*[*simp*]: *smap*·f·(xs :@ ys) = *smap*·f·xs :@ *smap*·f·ys
by (*induct xs*; *cases ys* = ⊥; *simp*)

lemma *stail-sappend*: *stail*·(xs :@ ys) = (*case xs of* [::] ⇒ *stail*·ys | z :# zs ⇒ zs :@ ys)
by (*induct xs*) *simp-all*

lemma *stail-append2*[*simp*]: xs ≠ [::] ⟹ *stail*·(xs :@ ys) = *stail*·xs :@ ys
by (*induct xs*) *simp-all*

lemma *slist-case-snoc*:
  g·⊥·⊥ = ⊥ ⟹ *slist-case*·f·g·(xs :@ [:x:]) = g·(*shead*·(xs :@ [:x:]))·(*stail*·(xs :@ [:x:]))
by (*cases x* = ⊥; *cases xs*; *clarsimp*)

fixrec *sall* :: ('a → *tr*) → [:'a:] → *tr* **where**
  sall·p·[::] = *TT*
| ⟦x ≠ ⊥; xs ≠ ⊥⟧ ⟹ *sall*·p·(x :# xs) = (p·x *andalso sall*·p·xs)

lemma *sall-strict*[*simp*]: *sall*·p·⊥ = ⊥
by *fixrec-simp*

lemma *sall-const-TT*[*simp*]:
  assumes xs ≠ ⊥
  shows *sall*·(Λ x. *TT*)·xs = *TT*
using *assms* **by** (*induct xs*) *simp-all*

lemma *sall-const-TT-conv*[*simp*]: (*sall*·(Λ x. *TT*)·xs = *TT*) ⟷ (xs ≠ ⊥)
by *auto*

lemma *sall-TT*[*simp*]: (*sall*·p·xs = *TT*) ⟷ (xs ≠ ⊥ ∧ (∀ x∈*sset xs*. p·x = *TT*))
by (*induct xs*) *simp-all*

fixrec *sfilter* :: ('a → *tr*) → [:'a:] → [:'a:] **where**
  sfilter·p·[::] = [::]
| ⟦x ≠ ⊥; xs ≠ ⊥⟧ ⟹ *sfilter*·p·(x :# xs) = *If* p·x *then* x :# *sfilter*·p·xs *else sfilter*·p·xs

lemma *sfilter-strict*[*simp*]: *sfilter*·p·⊥ = ⊥
by *fixrec-simp*

lemma *sfilter-bottom-iff*[*simp*]: (*sfilter*·p·xs = ⊥) ⟷ (xs = ⊥ ∨ (∃ x∈*sset xs*. p·x = ⊥))
by (*induct xs*) (*use trE* **in** *auto*)

lemma *sset-sfilter*[*simp*]:
  assumes ⋀x. x ∈ *sset xs* ⟹ p·x ≠ ⊥
  shows *sset* (*sfilter*·p·xs) = {x |x. x ∈ *sset xs* ∧ p·x = *TT*}

using *assms* **by** (*induct xs*) (*fastforce simp*: *If2-def*[*symmetric*] *split*: *If2-splits*)+

lemma *sfilter-strict-scons*[*simp*]:
  assumes $p\cdot\bot = \bot$
  shows *sfilter·p·*(*x* :# *xs*) = *If p·x then x* :# *sfilter·p·xs else sfilter·p·xs*
using *assms* **by** (*cases* $x = \bot$; *cases* $xs = \bot$; *simp*)

lemma *sfilter-scons-let*:
  assumes $p\cdot\bot = \bot$
  shows *sfilter·p·*(*x* :# *xs*) = (*let xs′* = *sfilter·p·xs in If p·x then x* :# *xs′ else xs′*)
unfolding *Let-def* **using** *assms* **by** *simp*

lemma *sfilter-sappend*[*simp*]: *sfilter·p·*(*xs* :@ *ys*) = *sfilter·p·xs* :@ *sfilter·p·ys*
by (*cases ys*; *clarsimp*) (*induct xs*; *fastforce simp*: *If2-def*[*symmetric*] *split*: *If2-splits*)

lemma *sfilter-const-FF*[*simp*]:
  assumes $xs \neq \bot$
  shows *sfilter·*(Λ *x*. *FF*)*·xs* = [::]
using *assms* **by** (*induct xs*) *simp-all*

lemma *sfilter-const-FF-conv*[*simp*]: (*sfilter·*(Λ *x*. *FF*)*·xs* = [::]) ⟷ ($xs \neq \bot$)
by *auto*

lemma *sfilter-const-TT*[*simp*]: *sfilter·*(Λ *x*. *TT*)*·xs* = *xs*
by (*induct xs*) *simp-all*

lemma *sfilter-cong*:
  assumes *xs* = *xs′*
  assumes $\bigwedge x.\ x \in sset\ xs \Longrightarrow p\cdot x = p'\cdot x$
  shows *sfilter·p·xs* = *sfilter·p′·xs′*
using *assms* **by** (*induct xs arbitrary*: *xs′*) *auto*

lemma *sfilter-snil-conv*[*simp*]: *sfilter·p·xs* = [::] ⟷ *sall·*(*neg oo p*)*·xs* = *TT*
by (*induct xs*; *force simp*: *If2-def*[*symmetric*] *split*: *If2-splits*)

lemma *sfilter-sfilter′*: *sfilter·p·*(*sfilter·q·xs*) = *sfilter·*(Λ *x*. *q·x andalso p·x*)*·xs*
proof(*induct xs*)
  case (*scons x xs*) **from** *scons*(*1*, *2*) **show** *?case*
    by (*cases* $sfilter\cdot q\cdot xs = \bot$)
      (*simp-all add*: *If-distr If-andalso scons*(*3*)[*symmetric*] *del*: *sfilter-bottom-iff*)
qed *simp-all*

lemma *sfilter-sfilter*: *sfilter·p oo sfilter·q* = *sfilter·*(Λ *x*. *q·x andalso p·x*)
by (*clarsimp simp*: *cfun-eq-iff sfilter-sfilter′*)

lemma *sfilter-smap′*:
  assumes $p\cdot\bot = \bot$
  shows *sfilter·p·*(*smap·f·xs*) = *smap·f·*(*sfilter·*(*p oo f*)*·xs*)
using *assms* **by** (*induct xs*; *simp add*: *If2-def*[*symmetric*] *split*: *If2-splits*) (*metis slist.con-rews*(*2*) *smap.simps*(*2*) *smap-strict*)

lemma *sfilter-smap*:
  assumes $p\cdot\bot = \bot$
  shows *sfilter·p oo smap·f* = *smap·f oo sfilter·*(*p oo f*)
using *assms* **by** (*clarsimp simp*: *cfun-eq-iff sfilter-smap′*)

fixrec *sfoldl* :: (′*a*::*pcpo* → ′*b*::*domain* → ′*a*) → ′*a* → [:′*b*:] → ′*a* **where**
  sfoldl·f·z·[::] = *z*

| $[\![x \neq \bot;\ xs \neq \bot]\!] \Longrightarrow sfoldl\cdot f\cdot z\cdot(x :\# \ xs) = sfoldl\cdot f\cdot(f\cdot z\cdot x)\cdot xs$

lemma *sfoldl-strict*[*simp*]: $sfoldl\cdot f\cdot z\cdot\bot = \bot$
by *fixrec-simp*

lemma *sfoldl-strict-f*[*simp*]:
  assumes $f\cdot\bot = \bot$
  shows $sfoldl\cdot f\cdot\bot\cdot xs = \bot$
using *assms* **by** (*induct xs*) *simp-all*

lemma *sfoldl-cong*:
  assumes $xs = xs'$
  assumes $z = z'$
  assumes $\bigwedge x\ z.\ x \in sset\ xs \Longrightarrow f\cdot z\cdot x = f'\cdot z\cdot x$
  shows $sfoldl\cdot f\cdot z\cdot xs = sfoldl\cdot f'\cdot z'\cdot xs'$
using *assms* **by** (*induct xs arbitrary*: $xs'\ z\ z'$) *auto*

lemma *sfoldl-sappend*[*simp*]:
  assumes $f\cdot\bot = \bot$
  shows $sfoldl\cdot f\cdot z\cdot(xs :@ \ ys) = sfoldl\cdot f\cdot(sfoldl\cdot f\cdot z\cdot xs)\cdot ys$
using *assms* **by** (*cases* $ys = \bot$, *force*) (*induct xs arbitrary*: *z*; *simp*)

fixrec *sfoldr* :: $('b \to 'a{::}pcpo \to 'a) \to 'a \to [:'b:] \to 'a$ **where**
  $sfoldr\cdot f\cdot z\cdot[::] = z$
| $[\![x \neq \bot;\ xs \neq \bot]\!] \Longrightarrow sfoldr\cdot f\cdot z\cdot(x :\# \ xs) = f\cdot x\cdot(sfoldr\cdot f\cdot z\cdot xs)$

lemma *sfoldr-strict*[*simp*]: $sfoldr\cdot f\cdot z\cdot\bot = \bot$
by *fixrec-simp*

fixrec *sconcat* :: $[:[:'a:]:] \to [:'a:]$ **where**
  $sconcat\cdot[::] = [::]$
| $[\![x \neq \bot;\ xs \neq \bot]\!] \Longrightarrow sconcat\cdot(x :\# \ xs) = x :@ \ sconcat\cdot xs$

lemma *sconcat-strict*[*simp*]: $sconcat\cdot\bot = \bot$
by *fixrec-simp*

lemma *sconcat-scons*[*simp*]:
  shows $sconcat\cdot(x :\# \ xs) = x :@ \ sconcat\cdot xs$
by (*cases* $x = \bot$, *force*) (*induct xs*; *fastforce*)

lemma *sconcat-sfoldl-aux*: $sfoldl\cdot sappend\cdot z\cdot xs = z :@ \ sconcat\cdot xs$
by (*induct xs arbitrary*: *z*) *simp-all*

lemma *sconcat-sfoldl*: $sconcat = sfoldl\cdot sappend\cdot[::]$
by (*clarsimp simp*: *cfun-eq-iff sconcat-sfoldl-aux*)

lemma *sconcat-sappend*[*simp*]: $sconcat\cdot(xs :@ \ ys) = sconcat\cdot xs :@ \ sconcat\cdot ys$
by (*induct xs*) *simp-all*

fixrec *slength* :: $[:'a:] \to Integer$
where
  $slength\cdot[::] = 0$
| $[\![x \neq \bot;\ xs \neq \bot]\!] \Longrightarrow slength\cdot(x :\# \ xs) = slength\cdot xs + 1$

lemma *slength-strict*[*simp*]: $slength\cdot\bot = \bot$
by *fixrec-simp*

lemma *slength-bottom-iff*[*simp*]: $(slength\cdot xs = \bot) \longleftrightarrow (xs = \bot)$

by (*induct xs*) *force*+

lemma *slength-ge-0*: *slength·xs* = *MkI·n* ⟹ *n* ≥ *0*
by (*induct xs arbitrary*: *n*) (*simp add*: *one-Integer-def plus-eq-MkI-conv*; *force*)+

lemma *slengthE*:
  shows ⟦*xs* ≠ ⊥; ⋀*n*. ⟦*slength·xs* = *MkI·n*; *0* ≤ *n*⟧ ⟹ *Q*⟧ ⟹ *Q*
by (*meson Integer.exhaust slength-bottom-iff slength-ge-0*)

lemma *slength-0-conv*[*simp*]:
  (*slength·xs* = *0*) ⟷ (*xs* = [::])
  (*slength·xs* = *MkI·0*) ⟷ (*xs* = [::])
  eq·0·(*slength·xs*) = *snull·xs*
  eq·(*slength·xs*)*·0* = *snull·xs*
by (*induct xs*) (*auto simp*: *one-Integer-def elim*: *slengthE*)

lemma *le-slength-0*[*simp*]: (*le·0·*(*slength·xs*) = *TT*) ⟷ (*xs* ≠ ⊥)
by (*cases slength·xs*) (*auto simp*: *slength-ge-0 zero-Integer-def*)

lemma *lt-slength-0*[*simp*]:
  xs ≠ ⊥ ⟹ *lt·*(*slength·xs*)*·0* = *FF*
  xs ≠ ⊥ ⟹ *lt·*(*slength·xs*)*·*(*slength·xs* + *1*) = *TT*
unfolding *zero-Integer-def one-Integer-def* **by** (*auto elim*: *slengthE*)

lemma *slength-smap*[*simp*]:
  assumes ⋀*x*. *x* ≠ ⊥ ⟹ *f·x* ≠ ⊥
  shows *slength·*(*smap·f·xs*) = *slength·xs*
using *assms* **by** (*induct xs*) *simp-all*

lemma *slength-sappend*[*simp*]: *slength·*(*xs* :@ *ys*) = *slength·xs* + *slength·ys*
by (*cases ys* = ⊥, *force*) (*induct xs*; *force simp*: *ac-simps*)

lemma *slength-sfoldl-aux*: *sfoldl·*(Λ *i* -. *i* + *1*)*·z·xs* = *z* + *slength·xs*
by (*induct xs arbitrary*: *z*) (*simp-all add*: *ac-simps*)

lemma *slength-sfoldl*: *slength* = *sfoldl·*(Λ *i* -. *i* + *1*)*·0*
by (*clarsimp simp*: *cfun-eq-iff slength-sfoldl-aux*)

lemma *le-slength-plus*:
  assumes *xs* ≠ ⊥
  assumes *n* ≠ ⊥
  shows *le·n·*(*slength·xs* + *n*) = *TT*
using *assms* **by** (*cases n*; *force elim*: *slengthE*)

fixrec *srev* :: [:'*a*:] → [:'*a*:] **where**
  srev·[::] = [::]
| ⟦*x* ≠ ⊥; *xs* ≠ ⊥⟧ ⟹ *srev·*(*x* :# *xs*) = *srev·xs* :@ [:*x*:]

lemma *srev-strict*[*simp*]: *srev·*⊥ = ⊥
by *fixrec-simp*

lemma *srev-bottom-iff*[*simp*]: (*srev·xs* = ⊥) ⟷ (*xs* = ⊥)
by (*induct xs*) *simp-all*

lemma *srev-scons*[*simp*]: *srev·*(*x* :# *xs*) = *srev·xs* :@ [:*x*:]
by (*cases x* = ⊥, *clarsimp*) (*induct xs*; *force*)

lemma *srev-sappend*[*simp*]: *srev·*(*xs* :@ *ys*) = *srev·ys* :@ *srev·xs*

by (*induct xs*) *simp-all*

lemma *srev-srev-ident*[*simp*]: $srev{\cdot}(srev{\cdot}xs) = xs$
by (*induct xs*) *auto*

lemma *srev-cases*[*case-names bottom snil ssnoc*]:
assumes $xs = \bot \Longrightarrow P$
assumes $xs = [::] \Longrightarrow P$
assumes $\bigwedge y\ ys.\ [\![y \neq \bot;\ ys \neq \bot;\ xs = ys\ :@\ [:y:]]\!] \Longrightarrow P$
shows P
using *assms* **by** (*metis slist.exhaust srev.simps*(*1*) *srev-scons srev-srev-ident srev-strict*)

lemma *srev-induct*[*case-names bottom snil ssnoc*]:
assumes $P\ \bot$
assumes $P\ [::]$
assumes $\bigwedge x\ xs.\ [\![x \neq \bot;\ xs \neq \bot;\ P\ xs]\!] \Longrightarrow P\ (xs\ :@\ [:x:])$
shows $P\ xs$
proof −
have $P\ (srev{\cdot}(srev{\cdot}xs))$ **by** (*rule slist.induct*[**where** $x{=}srev{\cdot}xs$]; *simp add*: *assms*)
then show *?thesis* **by** *simp*
qed

lemma *sfoldr-conv-sfoldl*:
assumes $\bigwedge x.\ f{\cdot}x{\cdot}\bot = \bot$ — *f* must be strict in the accumulator.
shows $sfoldr{\cdot}f{\cdot}z{\cdot}xs = sfoldl{\cdot}(\Lambda\ acc\ x.\ f{\cdot}x{\cdot}acc){\cdot}z{\cdot}(srev{\cdot}xs)$
using *assms* **by** (*induct xs arbitrary*: *z*) *simp-all*

fixrec *stake* :: $Integer \to [:'a:] \to [:'a:]$ **where** — Note: strict in both parameters.
$stake{\cdot}\bot{\cdot}\bot = \bot$
| $i \neq \bot \Longrightarrow stake{\cdot}i{\cdot}[::] = [::]$
| $[\![x \neq \bot;\ xs \neq \bot]\!] \Longrightarrow stake{\cdot}i{\cdot}(x :\#\ xs) = If\ le{\cdot}i{\cdot}0\ then\ [::]\ else\ x :\#\ stake{\cdot}(i - 1){\cdot}xs$

lemma *stake-strict*[*simp*]:
$stake{\cdot}\bot = \bot$
$stake{\cdot}i{\cdot}\bot = \bot$
by *fixrec-simp*+

lemma *stake-bottom-iff*[*simp*]: $(stake{\cdot}i{\cdot}xs = \bot) \longleftrightarrow (i = \bot \lor xs = \bot)$
by (*induct xs arbitrary*: *i*; *clarsimp*; *case-tac i*; *clarsimp*)

lemma *stake-0*[*simp*]:
$xs \neq \bot \Longrightarrow stake{\cdot}0{\cdot}xs = [::]$
$xs \neq \bot \Longrightarrow stake{\cdot}(MkI{\cdot}0){\cdot}xs = [::]$
$stake{\cdot}0{\cdot}xs \sqsubseteq [::]$
by (*cases xs*; *simp add*: *zero-Integer-def*)+

lemma *stake-scons*[*simp*]: $le{\cdot}1{\cdot}i = TT \Longrightarrow stake{\cdot}i{\cdot}(x :\#\ xs) = x :\#\ stake{\cdot}(i - 1){\cdot}xs$
by (*cases i*; *cases* $x = \bot$; *cases* $xs = \bot$;
simp add: *zero-Integer-def one-Integer-def split*: *if-splits*)

lemma *take-MkI-scons*[*simp*]:
$0 < n \Longrightarrow stake{\cdot}(MkI{\cdot}n){\cdot}(x :\#\ xs) = x :\#\ stake{\cdot}(MkI{\cdot}(n - 1)){\cdot}xs$
by (*cases* $x = \bot$; *cases* $xs = \bot$; *simp add*: *zero-Integer-def one-Integer-def*)

lemma *stake-numeral-scons*[*simp*]:
$xs \neq \bot \Longrightarrow stake{\cdot}1{\cdot}(x :\#\ xs) = [:x:]$
$stake{\cdot}(numeral\ (Num.Bit0\ k)){\cdot}(x :\#\ xs) = x :\#\ stake{\cdot}(numeral\ (Num.BitM\ k)){\cdot}xs$
$stake{\cdot}(numeral\ (Num.Bit1\ k)){\cdot}(x :\#\ xs) = x :\#\ stake{\cdot}(numeral\ (Num.Bit0\ k)){\cdot}xs$

by (*cases x* = ⊥; *cases xs*; *simp add*: *zero-Integer-def one-Integer-def numeral-Integer-eq*)+

lemma *stake-all*:
  assumes *le*·(*slength*·*xs*)·*i* = *TT*
  shows *stake*·*i*·*xs* = *xs*
using *assms*
proof(*induct xs arbitrary*: *i*)
  case (*scons x xs i*) **then show** *?case*
    by (*cases i*; *clarsimp simp*: *If2-def*[*symmetric*] *zero-Integer-def one-Integer-def split*: *If2-splits if-splits elim*!: *slengthE*)
qed (*simp-all add*: *le-defined*)

lemma *stake-all-triv*[*simp*]: *stake*·(*slength*·*xs*)·*xs* = *xs*
by (*cases xs* = ⊥) (*auto simp*: *stake-all*)

lemma *stake-append*[*simp*]: *stake*·*i*·(*xs* :@ *ys*) = *stake*·*i*·*xs* :@ *stake*·(*i* − *slength*·*xs*)·*ys*
proof(*induct xs arbitrary*: *i*)
  case (*snil i*) **then show** *?case* **by** (*cases i*; *simp add*: *zero-Integer-def*)
next
  case (*scons x xs i*) **then show** *?case*
    by (*cases i*; *cases ys*; *clarsimp simp*: *If2-def*[*symmetric*] *zero-Integer-def one-Integer-def split*: *If2-splits* *elim*!: *slengthE*)
qed *simp-all*

fixrec *sdrop* :: *Integer* → [:'*a*:] → [:'*a*:] **where** — Note: strict in both parameters.
  [*simp del*]: *sdrop*·*i*·*xs* = *If le*·*i*·*0 then xs else* (*case xs of* [::] ⇒ [::] | *y* :# *ys* ⇒ *sdrop*·(*i* − *1*)·*ys*)

lemma *sdrop-strict*[*simp*]:
  sdrop·⊥ = ⊥
  sdrop·*i*·⊥ = ⊥
by *fixrec-simp*+

lemma *sdrop-bottom-iff*[*simp*]: (*sdrop*·*i*·*xs* = ⊥) ⟷ (*i* = ⊥ ∨ *xs* = ⊥)
proof(*induct xs arbitrary*: *i*)
  case (*snil i*) **then show** *?case* **by** (*subst sdrop.unfold*) (*cases i*; *simp*)
next
  case (*scons x xs i*) **then show** *?case* **by** (*subst sdrop.unfold*) *fastforce*
qed *simp*

lemma *sdrop-snil*[*simp*]:
  assumes *i* ≠ ⊥
  shows *sdrop*·*i*·[::] = [::]
using *assms* **by** (*subst sdrop.unfold*; *fastforce*)

lemma *sdrop-snil-conv*[*simp*]: (*sdrop*·*i*·[::] = [::]) ⟷ (*i* ≠ ⊥)
by (*subst sdrop.unfold*; *fastforce*)

lemma *sdrop-0*[*simp*]:
  sdrop·*0*·*xs* = *xs*
  sdrop·(*MkI*·*0*)·*xs* = *xs*
by (*subst sdrop.simps*, *simp add*: *zero-Integer-def*)+

lemma *sdrop-pos*:
  le·*i*·*0* = *FF* ⟹ *sdrop*·*i*·*xs* = (*case xs of* [::] ⇒ [::] | *y* :# *ys* ⇒ *sdrop*·(*i* − *1*)·*ys*)
by (*subst sdrop.simps*, *simp*)

lemma *sdrop-neg*:
  le·*i*·*0* = *TT* ⟹ *sdrop*·*i*·*xs* = *xs*

by (*subst sdrop.simps*, *simp*)

lemma *sdrop-numeral-scons*[*simp*]:
  $x \neq \bot \Longrightarrow$ *sdrop*·*1*·(*x* :# *xs*) = *xs*
  $x \neq \bot \Longrightarrow$ *sdrop*·(*numeral* (*Num.Bit0 k*))·(*x* :# *xs*) = *sdrop*·(*numeral* (*Num.BitM k*))·*xs*
  $x \neq \bot \Longrightarrow$ *sdrop*·(*numeral* (*Num.Bit1 k*))·(*x* :# *xs*) = *sdrop*·(*numeral* (*Num.Bit0 k*))·*xs*
by (*subst sdrop.simps*,
    simp add: *zero-Integer-def one-Integer-def numeral-Integer-eq*; *cases xs*; *simp*)+

lemma *sdrop-sappend*[*simp*]:
  sdrop·*i*·(*xs* :@ *ys*) = *sdrop*·*i*·*xs* :@ *sdrop*·(*i* − *slength*·*xs*)·*ys*
proof(*induct xs arbitrary*: *i*)
  case (*snil i*) **then show** *?case* **by** (*cases i*; *simp add*: *zero-Integer-def*)
next
  case (*scons x xs i*) **then show** *?case*
    by (*cases ys* = ⊥; *cases le*·*i*·*0*; *cases i*;
        clarsimp simp: *zero-Integer-def one-Integer-def sdrop-neg sdrop-pos add.commute diff-diff-add*
            split: *if-splits elim*!: *slengthE*)
qed *simp*

lemma *sdrop-all*:
  assumes *le*·(*slength*·*xs*)·*i* = *TT*
  shows *sdrop*·*i*·*xs* = [::]
using *assms*
proof(*induct xs arbitrary*: *i*)
  case (*scons x xs i*) **then show** *?case*
    by (*subst sdrop.unfold*; *cases i*;
        clarsimp simp: *If2-def*[*symmetric*] *zero-Integer-def one-Integer-def split*: *If2-splits if-splits elim*!: *slengthE*)
qed (*simp-all add*: *le-defined*)

lemma *slength-sdrop*[*simp*]:
  slength·(*sdrop*·*i*·*xs*) = *If le*·*i*·*0 then slength*·*xs else If le*·(*slength*·*xs*)·*i then 0 else slength*·*xs* − *i*
proof(*induct xs arbitrary*: *i*)
  case (*snil i*) **then show** *?case* **by** (*cases i*; *simp add*: *zero-Integer-def*)
next
  case (*scons x xs i*) **then show** *?case*
    by (*subst sdrop.unfold*; *cases i*; *clarsimp simp*: *zero-Integer-def one-Integer-def elim*!: *slengthE*)
qed *simp*

lemma *sdrop-not-snilD*:
  assumes *sdrop*·(*MkI*·*i*)·*xs* ≠ [::]
  assumes *xs* ≠ ⊥
  shows *lt*·(*MkI*·*i*)·(*slength*·*xs*) = *TT* ∧ *xs* ≠ [::]
using *assms*
proof(*induct xs arbitrary*: *i*)
  case (*scons x xs i*) **then show** *?case*
   by (*subst* (*asm*) (*2*) *sdrop.unfold*, *clarsimp simp*: *zero-Integer-def one-Integer-def not-le sdrop-all elim*!: *slengthE*)
qed *simp-all*

lemma *sdrop-sappend-same*:
  assumes *xs* ≠ ⊥
  shows *sdrop*·(*slength*·*xs*)·(*xs* :@ *ys*) = *ys*
using *assms*
proof(*induct xs arbitrary*: *ys*)
  case (*scons x xs ys*) **then show** *?case*
    by (*cases ys* = ⊥; *subst sdrop.unfold*; *clarsimp simp*: *zero-Integer-def one-Integer-def elim*!: *slengthE*)
qed *simp-all*

fixrec *sscanl* :: $('a \to 'b \to 'a) \to 'a \to [:'b:] \to [:'a:]$ **where**
  $sscanl{\cdot}f{\cdot}z{\cdot}[::] = z :\# [::]$
| $[\![x \neq \bot;\ xs \neq \bot]\!] \Longrightarrow sscanl{\cdot}f{\cdot}z{\cdot}(x :\# xs) = z :\# sscanl{\cdot}f{\cdot}(f{\cdot}z{\cdot}x){\cdot}xs$

lemma *sscanl-strict*[*simp*]:
  $sscanl{\cdot}f{\cdot}\bot{\cdot}xs = \bot$
  $sscanl{\cdot}f{\cdot}z{\cdot}\bot = \bot$
by (*cases xs*) *fixrec-simp+*

lemma *sscanl-cong*:
  assumes $xs = xs'$
  assumes $z = z'$
  assumes $\bigwedge x\ z.\ x \in sset\ xs \Longrightarrow f{\cdot}z{\cdot}x = f'{\cdot}z{\cdot}x$
  shows $sscanl{\cdot}f{\cdot}z{\cdot}xs = sscanl{\cdot}f'{\cdot}z'{\cdot}xs'$
using *assms* **by** (*induct xs arbitrary*: $xs'\ z\ z'$) *auto*

lemma *sscanl-lfp-fusion'*:
  assumes $g{\cdot}\bot = \bot$
  assumes $*$: $\bigwedge acc\ x.\ x \neq \bot \Longrightarrow g{\cdot}(f{\cdot}acc{\cdot}x) = f'{\cdot}(g{\cdot}acc){\cdot}x$
  shows $smap{\cdot}g{\cdot}(sscanl{\cdot}f{\cdot}z{\cdot}xs) = sscanl{\cdot}f'{\cdot}(g{\cdot}z){\cdot}xs$
using *assms* **by** (*induct xs arbitrary*: *z*) *simp-all*

lemma *sscanl-lfp-fusion*:
  assumes $g{\cdot}\bot = \bot$
  assumes $*$: $\bigwedge acc\ x.\ x \neq \bot \Longrightarrow g{\cdot}(f{\cdot}acc{\cdot}x) = f'{\cdot}(g{\cdot}acc){\cdot}x$
  shows $smap{\cdot}g\ oo\ sscanl{\cdot}f{\cdot}z = sscanl{\cdot}f'{\cdot}(g{\cdot}z)$
using *assms* **by** (*clarsimp simp*: *cfun-eq-iff sscanl-lfp-fusion'*)

lemma *sscanl-ww-fusion'*: — Worker/wrapper (Gammie 2011; Gill and Hutton 2009) specialised to *sscanl*
  fixes $wrap :: 'b \to 'a$
  fixes $unwrap :: 'a \to 'b$
  fixes $z :: 'a$
  fixes $f :: 'a \to 'c \to 'a$
  fixes $f' :: 'b \to 'c \to 'b$
  assumes *ww*: $wrap\ oo\ unwrap = ID$
  assumes *wb*: $\bigwedge z\ x.\ x \neq \bot \Longrightarrow unwrap{\cdot}(f{\cdot}(wrap{\cdot}z){\cdot}x) = f'{\cdot}(unwrap{\cdot}(wrap{\cdot}z)){\cdot}x$
  shows $sscanl{\cdot}f{\cdot}z{\cdot}xs = smap{\cdot}wrap{\cdot}(sscanl{\cdot}f'{\cdot}(unwrap{\cdot}z){\cdot}xs)$
using *assms*
by (*induct xs arbitrary*: *z*) (*simp add*: *cfun-eq-iff retraction-cfcomp-strict* | *metis*)+

lemma *sscanl-ww-fusion*: — Worker/wrapper (Gammie 2011; Gill and Hutton 2009) specialised to *sscanl*
  fixes $wrap :: 'b \to 'a$
  fixes $unwrap :: 'a \to 'b$
  fixes $z :: 'a$
  fixes $f :: 'a \to 'c \to 'a$
  fixes $f' :: 'b \to 'c \to 'b$
  assumes *ww*: $wrap\ oo\ unwrap = ID$
  assumes *wb*: $\bigwedge z\ x.\ x \neq \bot \Longrightarrow unwrap{\cdot}(f{\cdot}(wrap{\cdot}z){\cdot}x) = f'{\cdot}(unwrap{\cdot}(wrap{\cdot}z)){\cdot}x$
  shows $sscanl{\cdot}f{\cdot}z = smap{\cdot}wrap\ oo\ sscanl{\cdot}f'{\cdot}(unwrap{\cdot}z)$
using *assms* **by** (*clarsimp simp*: *cfun-eq-iff sscanl-ww-fusion'*)

fixrec *sinits* :: $[:'a:] \to [:[:'a:]:]$ **where**
  $sinits{\cdot}[::] = [::] :\# [::]$
| $[\![x \neq \bot;\ xs \neq \bot]\!] \Longrightarrow sinits{\cdot}(x :\# xs) = [::] :\# smap{\cdot}(scons{\cdot}x){\cdot}(sinits{\cdot}xs)$

lemma *sinits-strict*[*simp*]: $sinits{\cdot}\bot = \bot$
by *fixrec-simp*

lemma *sinits-bottom-iff* [*simp*]: (*sinits*·*xs* = ⊥) ⟷ (*xs* = ⊥)
by (*induct xs*) *simp-all*

lemma *sinits-not-snil*[*iff*]: *sinits*·*xs* ≠ [::]
by (*cases xs*) *simp-all*

lemma *sinits-empty-bottom*[*simp*]: (*sset* (*sinits*·*xs*) = {}) ⟷ (*xs* = ⊥)
by (*cases xs*) *simp-all*

lemma *sinits-scons*[*simp*]: *sinits*·(*x* :# *xs*) = [::] :# *smap*·(*x* :#)·(*sinits*·*xs*)
by (*cases x* = ⊥, *force*) (*induct xs*; *force*)

lemma *sinits-length*[*simp*]: *slength*·(*sinits*·*xs*) = *slength*·*xs* + *1*
by (*induct xs*) *simp-all*

lemma *sinits-snoc*[*simp*]: *sinits*·(*xs* :@ [:*x*:]) = *sinits*·*xs* :@ [:*xs* :@ [:*x*:]:]
by (*induct xs*) *simp-all*

lemma *sinits-foldr′*: — Bird (1987, p30)
shows *sinits*·*xs* = *sfoldr*·(Λ *x xs*. [:[::]:] :@ *smap*·(*x* :#)·*xs*)·[:[::]:]·*xs*
by (*induct xs*) *simp-all*

lemma *sinits-sscanl′*:
shows *smap*·(*sfoldl*·*f*·*z*)·(*sinits*·*xs*) = *sscanl*·*f*·*z*·*xs*
by (*induct xs arbitrary*: *z*) (*simp-all cong*: *smap-cong add*: *oo-def eta-cfun*)

lemma *sinits-sscanl*: — Bird (1987, Lemma 5), Bird (2010, p118 "the scan lemma")
shows *smap*·(*sfoldl*·*f*·*z*) *oo sinits* = *sscanl*·*f*·*z*
by (*simp add*: *sinits-sscanl′ cfun-eq-iff*)

lemma *sinits-all*[*simp*]: (*xs* ∈ *sset* (*sinits*·*xs*)) ⟷ (*xs* ≠ ⊥)
by (*induct xs*) *simp-all*

fixrec *stails* :: [:′*a*:] → [:[:′*a*:]:] **where**
stails·[::] = [::] :# [::]
| ⟦*x* ≠ ⊥; *xs* ≠ ⊥⟧ ⟹ *stails*·(*x* :# *xs*) = (*x* :# *xs*) :# *stails*·*xs*

lemma *stails-strict*[*simp*]: *stails*·⊥ = ⊥
by *fixrec-simp*

lemma *stails-bottom-iff* [*simp*]: (*stails*·*xs* = ⊥) ⟷ (*xs* = ⊥)
by (*induct xs*) *simp-all*

lemma *stails-not-snil*[*iff*]: *stails*·*xs* ≠ [::]
by (*cases xs*) *simp-all*

lemma *stails-scons*[*simp*]: *stails*·(*x* :# *xs*) = (*x* :# *xs*) :# *stails*·*xs*
by (*induct xs*) (*cases x* = ⊥; *simp*)+

lemma *stails-slength*[*simp*]: *slength*·(*stails*·*xs*) = *slength*·*xs* + *1*
by (*induct xs*) *simp-all*

lemma *stails-snoc*[*simp*]:
shows *stails*·(*xs* :@ [:*x*:]) = *smap*·(Λ *ys*. *ys* :@ [:*x*:])·(*stails*·*xs*) :@ [:[::]:]
by (*induct xs*) *simp-all*

lemma *stails-sfoldl′*:
shows *stails*·*xs* = *sfoldl*·(Λ *xs x*. *smap*·(Λ *ys*. *ys* :@ [:*x*:])·*xs* :@ [:[::]:])·[:[::]:]·*xs*

by (*induct xs rule*: *srev-induct*) *simp-all*

lemma *stails-sfoldl*:
  shows $stails = sfoldl\cdot(\Lambda\ xs\ x.\ smap\cdot(\Lambda\ ys.\ ys\ {:}@\ [{:}x{:}])\cdot xs\ {:}@\ [{:}[{::}]{:}])\cdot[{:}[{::}]{:}]$
by (*clarsimp simp*: *cfun-eq-iff stails-sfoldl′*)

lemma *stails-all*[*simp*]: $(xs \in sset\ (stails\cdot xs)) \longleftrightarrow (xs \neq \bot)$
by (*cases xs*) *simp-all*

fixrec *selem* :: $'a{::}Eq\text{-}def \rightarrow [{:}'a{:}] \rightarrow tr$ **where**
  $selem\cdot x\cdot[{::}] = FF$
| $[\![y \neq \bot;\ ys \neq \bot]\!] \Longrightarrow selem\cdot x\cdot(y\ {:}\#\ ys) = (eq\cdot x\cdot y\ orelse\ selem\cdot x\cdot ys)$

lemma *selem-strict*[*simp*]: $selem\cdot x\cdot\bot = \bot$
by *fixrec-simp*

lemma *selem-bottom-iff*[*simp*]: $(selem\cdot x\cdot xs = \bot) \longleftrightarrow (xs = \bot \vee (xs \neq [{::}] \wedge x = \bot))$
by (*induct xs*) *auto*

lemma *selem-sappend*[*simp*]:
  assumes $ys \neq \bot$
  shows $selem\cdot x\cdot(xs\ {:}@\ ys) = (selem\cdot x\cdot xs\ orelse\ selem\cdot x\cdot ys)$
using *assms* **by** (*induct xs*) *simp-all*

lemma *elem-TT*[*simp*]: $(selem\cdot x\cdot xs = TT) \longleftrightarrow (x \in sset\ xs)$
by (*induct xs*; *auto*) (*metis sset-defined*)+

lemma *elem-FF*[*simp*]: $(selem\cdot x\cdot xs = FF) \longleftrightarrow (xs = [{::}] \vee (x \neq \bot \wedge xs \neq \bot \wedge x \notin sset\ xs))$
by (*induct xs*) *auto*

lemma *selem-snil-stails*[*iff*]:
  assumes $xs \neq \bot$
  shows $selem\cdot[{::}]\cdot(stails\cdot xs) = TT$
using *assms* **by** (*induct xs*) *simp-all*

fixrec *sconcatMap* :: $('a \rightarrow [{:}'b{:}]) \rightarrow [{:}'a{:}] \rightarrow [{:}'b{:}]$ **where**
[*simp del*]: $sconcatMap\cdot f = sconcat\ oo\ smap\cdot f$

lemma *sconcatMap-strict*[*simp*]: $sconcatMap\cdot f\cdot\bot = \bot$
by *fixrec-simp*

lemma *sconcatMap-snil*[*simp*]: $sconcatMap\cdot f\cdot[{::}] = [{::}]$
by *fixrec-simp*

lemma *sconcatMap-scons*[*simp*]: $x \neq \bot \Longrightarrow sconcatMap\cdot f\cdot(x\ {:}\#\ xs) = f\cdot x\ {:}@\ sconcatMap\cdot f\cdot xs$
by (*cases* $xs = \bot$; *simp add*: *sconcatMap.unfold*)

lemma *sconcatMap-bottom-iff*[*simp*]: $(sconcatMap\cdot f\cdot xs = \bot) \longleftrightarrow (xs = \bot \vee (\exists x{\in}sset\ xs.\ f\cdot x = \bot))$
by (*induct xs*) *simp-all*

lemma *sconcatMap-sappend*[*simp*]: $sconcatMap\cdot f\cdot(xs\ {:}@\ ys) = sconcatMap\cdot f\cdot xs\ {:}@\ sconcatMap\cdot f\cdot ys$
by (*induct xs*) *simp-all*

lemma *sconcatMap-monad-laws*:
  $sconcatMap\cdot(\Lambda\ x.\ [{:}x{:}])\cdot xs = xs$
  $sconcatMap\cdot g\cdot(sconcatMap\cdot f\cdot xs) = sconcatMap\cdot(\Lambda\ x.\ sconcatMap\cdot g\cdot(f\cdot x))\cdot xs$
by (*induct xs*) *simp-all*

```
fixrec supto :: Integer → Integer → [:Integer:] where
  [simp del]: supto·i·j = If le·i·j then i :# supto·(i+1)·j else [::]

lemma upto-strict[simp]:
  supto·⊥ = ⊥
  supto·m·⊥ = ⊥
by fixrec-simp+

lemma supto-is-snil-conv[simp]:
  (supto·(MkI·i)·(MkI·j) = [::]) ⟷ (j < i)
  ([::] = supto·(MkI·i)·(MkI·j)) ⟷ (j < i)
by (subst supto.unfold; simp)+

lemma supto-simp[simp]:
  j < i ⟹ supto·(MkI·i)·(MkI·j) = [::]
  i ≤ j ⟹ supto·(MkI·i)·(MkI·j) = MkI·i :# supto·(MkI·i+1)·(MkI·j)
  supto·0·0 = [:0:]
by (subst supto.simps, simp)+

lemma supto-defined[simp]: supto·(MkI·i)·(MkI·j) ≠ ⊥ (is ?P i j)
proof (cases j − i)
  fix d
  assume j − i = int d
  then show ?P i j
  proof (induct d arbitrary: i j)
    case (Suc d i j)
    then have j − (i + 1) = int d and le: i ≤ j by simp-all
    from Suc(1)[OF this(1)] have IH: ?P (i+1) j .
    then show ?case using le by (simp add: one-Integer-def)
  qed (simp add: one-Integer-def)
next
  fix d
  assume j − i = − int d
  then have j ≤ i by auto
  moreover
  { assume j = i then have ?P i j by (simp add: one-Integer-def) }
  moreover
  { assume j < i then have ?P i j by (simp add: one-Integer-def) }
  ultimately show ?thesis by arith
qed

lemma supto-bottom-iff[simp]:
  (supto·i·j = ⊥) ⟷ (i = ⊥ ∨ j = ⊥)
by (cases i; simp; cases j; simp)

lemma supto-snoc[simp]:
  i ≤ j ⟹ supto·(MkI·i)·(MkI·j) = supto·(MkI·i)·(MkI·j−1) :@ [:MkI·j:]
proof(induct nat(j − i) arbitrary: i j)
  case 0 then show ?case by (simp add: one-Integer-def)
next
  case (Suc k i j)
  then have k = nat (j − (i + 1)) i < j by linarith+
  from this(2) Suc.hyps(1)[OF this(1)] Suc(2,3) show ?case by (simp add: one-Integer-def)
qed

lemma slength-supto[simp]: slength·(supto·(MkI·i)·(MkI·j)) = MkI·(if j < i then 0 else j − i + 1) (is ?P i j)
proof (cases j − i)
  fix d
```

```
  assume j − i = int d
  then show ?P i j
  proof (induct d arbitrary: i j)
    case (Suc d i j)
    then have j − (i + 1) = int d and le: i ≤ j by simp-all
    from Suc(1)[OF this(1)] have IH: ?P (i+1) j .
    then show ?case using le by (simp add: one-Integer-def)
  qed (simp add: one-Integer-def)
next
  fix d
  assume j − i = − int d
  then have j ≤ i by auto
  moreover
  { assume j = i then have ?P i j by (simp add: one-Integer-def) }
  moreover
  { assume j < i then have ?P i j by (simp add: one-Integer-def) }
  ultimately show ?thesis by arith
qed

lemma sset-supto[simp]:
  sset (supto·(MkI·i)·(MkI·j)) = {MkI·k |k. i ≤ k ∧ k ≤ j} (is sset (?u i j) = ?R i j)
proof (cases j − i)
  case (nonneg k)
  then show ?thesis
  proof (induct k arbitrary: i j)
    case (Suc k)
    then have ∗: j − (i + 1) = int k by simp
    from Suc(1)[OF ∗] have IH: sset (?u (i+1) j) = ?R (i+1) j .
    from ∗ have i ≤ j by simp
    then have sset (?u i j) = sset (MkI·i :# ?u (i+1) j) by (simp add: one-Integer-def)
    also have ... = insert (MkI·i) (?R (i+1) j) by (simp add: IH)
    also have ... = ?R i j using ‹i ≤ j› by auto
    finally show ?case .
  qed (force simp: one-Integer-def)
qed simp

lemma supto-split1: — From HOL.List
  assumes i ≤ j
  assumes j ≤ k
  shows supto·(MkI·i)·(MkI·k) = supto·(MkI·i)·(MkI·(j − 1)) :@ supto·(MkI·j)·(MkI·k)
using assms
proof (induct j rule: int-ge-induct)
  case (step l) with supto-simp(2) supto-snoc show ?case by (clarsimp simp: one-Integer-def)
qed simp

lemma supto-split2: — From HOL.List
  assumes i ≤ j
  assumes j ≤ k
  shows supto·(MkI·i)·(MkI·k) = supto·(MkI·i)·(MkI·j) :@ supto·(MkI·(j + 1))·(MkI·k)
proof(cases j + 1 ≤ k)
  case True with assms show ?thesis
    by (subst supto-split1[where j=j + 1 and k=k]; clarsimp simp: one-Integer-def)
next
  case False with assms show ?thesis by (clarsimp simp: one-Integer-def not-le)
qed

lemma supto-split3: — From HOL.List
  assumes i ≤ j
```

assumes $j \leq k$
shows *supto*·(*MkI*·*i*)·(*MkI*·*k*) = *supto*·(*MkI*·*i*)·(*MkI*·(*j* − *1*)) :@ *MkI*·*j* :# *supto*·(*MkI*·(*j* + *1*))·(*MkI*·*k*)
using *assms supto-simp*(*2*) *supto-split1* **by** (*metis one-Integer-def plus-MkI-MkI*)

lemma *sinits-stake′*:
shows *sinits*·*xs* = *smap*·(Λ *i*. *stake*·*i*·*xs*)·(*supto*·*0*·(*slength*·*xs*))
proof(*induct xs rule*: *srev-induct*)
case (*ssnoc x xs*) **then show** *?case*
apply (*clarsimp simp*: *zero-Integer-def one-Integer-def stake-all*
simp del: *supto-simp*
elim!: *slengthE*)
apply (*rule arg-cong*, *rule smap-cong*[*OF refl*])
apply *clarsimp*
done
qed *simp-all*

lemma *stails-sdrop′*:
shows *stails*·*xs* = *smap*·(Λ *i*. *sdrop*·*i*·*xs*)·(*supto*·*0*·(*slength*·*xs*))
proof(*induct xs rule*: *srev-induct*)
case (*ssnoc x xs*) **then show** *?case*
apply (*clarsimp simp*: *zero-Integer-def one-Integer-def sdrop-all*
simp del: *supto-simp*
elim!: *slengthE*)
apply (*rule arg-cong*, *rule smap-cong*[*OF refl*])
apply *clarsimp*
apply (*subst* (*3*) *sdrop-neg*; *fastforce simp*: *zero-Integer-def*)
done
qed *simp-all*

lemma *sdrop-elem-stails*[*iff*]:
assumes $xs \neq \bot$
shows *sdrop*·(*MkI*·*i*)·*xs* ∈ *sset* (*stails*·*xs*)
using *assms*
by (*clarsimp simp*: *stails-sdrop′ zero-Integer-def one-Integer-def elim*!: *slengthE*)
(*metis add.left-neutral le-MkI-MkI le-cases not-less sdrop-all sdrop-neg zero-Integer-def zless-imp-add1-zle*)

fixrec *slast* :: [:′*a*:] → ′*a* **where**
slast·[::] = ⊥
| ⟦*x* ≠ ⊥; *xs* ≠ ⊥⟧ ⟹ *slast*·(*x* :# *xs*) = (*case xs of* [::] ⇒ *x* | *y* :# *ys* ⇒ *slast*·*xs*)

lemma *slast-strict*[*simp*]:
slast·⊥ = ⊥
by *fixrec-simp*

lemma *slast-singleton*[*simp*]: *slast*·[:*x*:] = *x*
by (*cases x* = ⊥; *simp*)

lemma *slast-sappend-ssnoc*[*simp*]:
assumes $xs \neq \bot$
shows *slast*·(*xs* :@ [:*x*:]) = *x*
using *assms*
proof(*induct xs*)
case (*scons y ys*) **then show** *?case* **by** (*cases x* = ⊥; *simp*; *cases ys*; *simp*)
qed *simp-all*

fixrec *sbutlast* :: [:′*a*:] → [:′*a*:] **where**
sbutlast·[::] = [::]
| ⟦*x* ≠ ⊥; *xs* ≠ ⊥⟧ ⟹ *sbutlast*·(*x* :# *xs*) = (*case xs of* [::] ⇒ [::] | *y* :# *ys* ⇒ *x* :# *sbutlast*·*xs*)

lemma *sbutlast-strict*[*simp*]:
  sbutlast·⊥ = ⊥
by *fixrec-simp*

lemma *sbutlast-sappend-ssnoc*[*simp*]:
  assumes *x* ≠ ⊥
  shows *sbutlast*·(*xs* :@ [:*x*:]) = *xs*
using *assms*
proof(*induct xs*)
  case (*scons y ys*) **then show** *?case* **by** (*cases ys*; *simp*)
qed *simp-all*

fixrec *prefix* :: [:′*a*::*Eq-def*:] → [:′*a*:] → *tr* **where**
  prefix·*xs*·⊥ = ⊥
| *ys* ≠ ⊥ ⟹ *prefix*·[::]·*ys* = *TT*
| ⟦*x* ≠ ⊥; *xs* ≠ ⊥⟧ ⟹ *prefix*·(*x* :# *xs*)·[::] = *FF*
| ⟦*x* ≠ ⊥; *xs* ≠ ⊥; *y* ≠ ⊥; *ys* ≠ ⊥⟧ ⟹ *prefix*·(*x* :# *xs*)·(*y* :# *ys*) = (*eq*·*x*·*y* *andalso* *prefix*·*xs*·*ys*)

lemma *prefix-strict*[*simp*]: *prefix*·⊥ = ⊥
by (*clarsimp simp*: *cfun-eq-iff*) *fixrec-simp*

lemma *prefix-bottom-iff*[*simp*]: (*prefix*·*xs*·*ys* = ⊥) ⟷ (*xs* = ⊥ ∨ *ys* = ⊥)
proof(*induct xs arbitrary*: *ys*)
  case (*snil ys*) **then show** *?case* **by** (*cases ys*) *simp-all*
next
  case (*scons a xs*) **then show** *?case* **by** (*cases ys*) *simp-all*
qed *simp*

lemma *prefix-definedD*:
  assumes *prefix*·*xs*·*ys* = *TT*
  shows *xs* ≠ ⊥ ∧ *ys* ≠ ⊥
using *assms* **by** (*induct xs arbitrary*: *ys*) *auto*

lemma *prefix-refl*[*simp*]:
  assumes *xs* ≠ ⊥
  shows *prefix*·*xs*·*xs* = *TT*
using *assms* **by** (*induct xs*) *simp-all*

lemma *prefix-refl-conv*[*simp*]: (*prefix*·*xs*·*xs* = *TT*) ⟷ (*xs* ≠ ⊥)
by *auto*

lemma *prefix-of-snil*[*simp*]: *prefix*·*xs*·[::] = (*case xs of* [::] ⇒ *TT* | *x* :# *xs* ⇒ *FF*)
by (*cases xs*) *simp-all*

lemma *prefix-singleton-TT*:
  shows *prefix*·[:*x*:]·*ys* = *TT* ⟷ (*x* ≠ ⊥ ∧ (∃ *zs*. *zs* ≠ ⊥ ∧ *ys* = *x* :# *zs*))
by (*cases x* = ⊥; *clarsimp*; *cases ys*; *fastforce*)

lemma *prefix-singleton-FF*:
  shows *prefix*·[:*x*:]·*ys* = *FF* ⟷ (*x* ≠ ⊥ ∧ (*ys* = [::] ∨ (∃ *z zs*. *z* ≠ ⊥ ∧ *zs* ≠ ⊥ ∧ *ys* = *z* :# *zs* ∧ *x* ≠ *z*)))
by (*cases x* = ⊥; *clarsimp*; *cases ys*; *fastforce*)

lemma *prefix-FF-not-snilD*:
  assumes *prefix*·*xs*·*ys* = *FF*
  shows *xs* ≠ [::]
using *assms* **by** (*cases xs*; *cases ys*; *simp*)

lemma *prefix-slength*:
  assumes *prefix·xs·ys* = *TT*
  shows *le*·(*slength*·*xs*)·(*slength*·*ys*) = *TT*
using *assms*
proof(*induct ys arbitrary*: *xs*)
  case (*snil xs*) **then show** *?case* **by** (*cases xs*) *simp-all*
next
  case (*scons a ys*) **then show** *?case* **by** (*cases xs*) (*simp-all add*: *le-plus-1*)
qed *simp*

lemma *prefix-slength-strengthen*: *prefix·xs·ys* = (*le*·(*slength*·*xs*)·(*slength*·*ys*) *andalso prefix·xs·ys*)
by (*rule andalso-weaken-left*) (*auto dest*: *prefix-slength*)

lemma *prefix-scons-snil*[*simp*]: *prefix*·(*x* :# *xs*)·[::] ≠ *TT*
by (*cases x* :# *xs* ≠ ⊥) *auto*

lemma *scons-prefix-scons*[*simp*]:
  (*prefix*·(*x* :# *xs*)·(*y* :# *ys*) = *TT*) ⟷ (*eq*·*x*·*y* = *TT* ∧ *prefix·xs·ys* = *TT*)
by (*cases x* :# *xs* ≠ ⊥ ∧ *y* :# *ys* ≠ ⊥) *auto*

lemma *append-prefixD*:
  assumes *prefix*·(*xs* :@ *ys*)·*zs* = *TT*
  shows *prefix·xs·zs* = *TT*
using *assms*
proof(*induct xs arbitrary*: *zs*)
  case (*snil zs*) **then show** *?case* **using** *prefix.simps*(*2*) **by** *force*
next
  case (*scons x xs zs*) **then show** *?case*
    by (*metis prefix.simps*(*1*) *prefix-scons-snil sappend-scons scons-prefix-scons slist.exhaust*)
qed *simp*

lemma *same-prefix-prefix*[*simp*]:
  assumes *xs* ≠ ⊥
  shows *prefix*·(*xs* :@ *ys*)·(*xs* :@ *zs*) = *prefix·ys·zs*
using *assms*
proof(*cases ys* = ⊥ *zs* = ⊥ *rule*: *bool.exhaust*[*case-product bool.exhaust*])
  case *False-False* **with** *assms* **show** *?thesis* **by** (*induct xs*) *simp-all*
qed *simp-all*

lemma *eq-prefix-TT*:
  assumes *eq·xs·ys* = *TT*
  shows *prefix·xs·ys* = *TT*
using *assms* **by** (*induct xs arbitrary*: *ys*) (*case-tac ys*; *simp*)+

lemma *prefix-eq-FF*:
  assumes *prefix·xs·ys* = *FF*
  shows *eq·xs·ys* = *FF*
using *assms* **by** (*induct xs arbitrary*: *ys*) (*case-tac ys*; *auto*)+

lemma *prefix-slength-eq*:
  shows *eq·xs·ys* = (*eq*·(*slength*·*xs*)·(*slength*·*ys*) *andalso prefix·xs·ys*)
proof(*induct xs arbitrary*: *ys*)
  case (*snil ys*) **then show** *?case*
    by (*cases ys*; *clarsimp simp*: *one-Integer-def elim*!: *slengthE*)
next
  case (*scons x xs ys*) **then show** *?case*
    by (*cases ys*; *clarsimp simp*: *zero-Integer-def one-Integer-def elim*!: *slengthE*)
qed *simp*

```
lemma stake-slength-plus-1:
  shows stake·(slength·xs + 1)·(y :# ys) = y :# stake·(slength·xs)·ys
by (cases xs = ⊥ y = ⊥ ys = ⊥ rule: bool.exhaust[case-product bool.exhaust bool.exhaust]; clarsimp)
   (auto simp: If2-def[symmetric] zero-Integer-def one-Integer-def split: If2-splits elim!: slengthE)

lemma sdrop-slength-plus-1:
  assumes y ≠ ⊥
  shows sdrop·(slength·xs + 1)·(y :# ys) = sdrop·(slength·xs)·ys
using assms
by (subst sdrop.simps;
    cases xs = ⊥; clarsimp; cases ys = ⊥;
    clarsimp simp: If2-def[symmetric] zero-Integer-def one-Integer-def split: If2-splits elim!: slengthE)

lemma eq-take-length-prefix: prefix·xs·ys = eq·xs·(stake·(slength·xs)·ys)
proof (induct xs arbitrary: ys)
  case (snil ys) show ?case by (cases ys; clarsimp)
next
  case (scons x xs ys)
  note IH = this
  show ?case
  proof (cases slength·xs = ⊥)
    case True then show ?thesis by simp
  next
    case False
    show ?thesis
    proof (cases ys)
      case bottom
      then show ?thesis using False
        using le-slength-plus[of xs 1] by simp
    next
      case snil then show ?thesis using False and IH(1,2) by simp
    next
      case (scons z zs)
      then show ?thesis
        using False and IH(1,2) IH(3)[of zs]
        by (simp add: stake-slength-plus-1 monofun-cfun-arg)
    qed
  qed
qed simp

lemma prefix-sdrop-slength:
  assumes prefix·xs·ys = TT
  shows xs :@ sdrop·(slength·xs)·ys = ys
using assms by (induct xs arbitrary: ys) (case-tac ys; simp add: sdrop-slength-plus-1)+

lemma prefix-sdrop-prefix-eq:
  assumes prefix·xs·ys = TT
  shows eq·(sdrop·(slength·xs)·ys)·[::] = eq·ys·xs
using assms by (induct xs arbitrary: ys) (case-tac ys; simp add: sdrop-slength-plus-1)+
```

4 Knuth-Morris-Pratt matching according to Bird

4.1 Step 1: Specification

We begin with the specification of string matching given by Bird (2010, Chapter 16). (References to "Bird" in the following are to this text.) Note that we assume *eq* has some nice properties (see §2.2) and use strict lists.

fixrec *endswith* :: [:'*a*::*Eq-def*:] → [:'*a*:] → *tr* **where**
[*simp del*]: *endswith·pat* = *selem·pat oo stails*

fixrec *matches* :: [:'*a*::*Eq-def*:] → [:'*a*:] → [:*Integer*:] **where**
[*simp del*]: *matches·pat* = *smap·slength oo sfilter·*(*endswith·pat*) *oo sinits*

Bird describes *matches·pat·xs* as returning "a list of integers *p* such that *pat* is a suffix of *stake·p·xs*." The following examples illustrate this behaviour:

lemma *matches*·[::]·[::] = [:*0*:]
unfolding *matches.unfold endswith.unfold* **by** *simp*

lemma *matches*·[::]·[:*10*::*Integer*, *20*, *30*:] = [:*0*, *1*, *2*, *3*:]
unfolding *matches.unfold endswith.unfold* **by** *simp*

lemma *matches*·[:*1*::*Integer*,*2*,*3*,*1*,*2*:]·[:*1*,*2*,*1*,*2*,*3*,*1*,*2*,*3*,*1*,*2*:] = [:*7*, *10*:]
unfolding *matches.unfold endswith.unfold*
by (*simp add*: *sfilter-scons-let del*: *sfilter-strict-scons sfilter.simps*)

lemma *endswith-strict*[*simp*]:
  endswith·⊥ = ⊥
  endswith·pat·⊥ = ⊥
by (*fixrec-simp*; *simp add*: *cfun-eq-iff*)+

lemma *matches-strict*[*simp*]:
  matches·⊥ = ⊥
  matches·pat·⊥ = ⊥
by (*fixrec-simp*; *clarsimp simp*: *cfun-eq-iff*)+

Bird's strategy for deriving KMP from this specification is encoded in the following lemmas: if we can rewrite *endswith* as a composition of a predicate with a *sfoldl*, then we can rewrite *matches* into a *sscanl*.

lemma *fork-sfoldl*:
  shows *sfoldl·f1·z1* && *sfoldl·f2·z2* = *sfoldl*·(Λ (*a*, *b*) *z*. (*f1·a·z*, *f2·b·z*))·(*z1*, *z2*) (**is** *?lhs* = *?rhs*)
proof(*rule cfun-eqI*)
  fix *xs* **show** *?lhs·xs* = *?rhs·xs*
    by (*induct xs arbitrary*: *z1 z2*) *simp-all*
qed

lemma *smap-sfilter-split-cfcomp*: — Bird (16.4)
  assumes *f*·⊥ = ⊥
  assumes *p*·⊥ = ⊥
  shows *smap·f oo sfilter*·(*p oo g*) = *smap·cfst oo sfilter*·(*p oo csnd*) *oo smap*·(*f* && *g*) (**is** *?lhs* = *?rhs*)
proof(*rule cfun-eqI*)
  fix *xs* **show** *?lhs·xs* = *?rhs·xs*
    using *assms* **by** (*induct xs*) (*simp-all add*: *If2-def*[*symmetric*] *split*: *If2-splits*)
qed

lemma *Bird-strategy*:
  assumes *endswith*: *endswith·pat* = *p oo sfoldl·op·z*
  assumes *step*: *step* = (Λ (*n*, *x*) *y*. (*n* + *1*, *op·x·y*))
  assumes *p*·⊥ = ⊥ — We can reasonably expect the predicate to be strict
  shows *matches·pat* = *smap·cfst oo sfilter*·(*p oo csnd*) *oo sscanl·step*·(*0*, *z*)
apply (*simp add*: *matches.simps assoc-oo endswith*)
apply (*subst smap-sfilter-split-cfcomp*, *fastforce*, *fact*)
apply (*subst slength-sfoldl*)
apply (*subst fork-sfoldl*)
apply (*simp add*: *oo-assoc*[*symmetric*])
apply (*subst sinits-sscanl*)
apply (*fold step*)

apply (*rule refl*)
done

Bird proceeds by reworking *endswith* into the form required by *Bird-strategy*. This is eased by an alternative definition of *endswith*.

lemma *sfilter-supto*:
  assumes *0 ≤ d*
  shows *sfilter·(Λ x. le·(MkI·n − x)·(MkI·d))·(supto·(MkI·m)·(MkI·n))*
    = supto·(MkI·(if m ≤ n − d then n − d else m))·(MkI·n) (**is** *?sfilterp·?suptomn = -*)
proof(*cases m ≤ n − d*)
  case *True*
  moreover
  from *True assms* **have** *?sfilterp·?suptomn = ?sfilterp·(supto·(MkI·m)·(MkI·(n − d − 1)) :@ supto·(MkI·(n − d))·(MkI·n))*
    using *supto-split1* **by** *auto*
  moreover from *True assms* **have** *?sfilterp·(supto·(MkI·m)·(MkI·(n − d − 1))) = [::]* **by** *auto*
  ultimately show *?thesis* **by** (*clarsimp intro*!: *trans*[*OF sfilter-cong*[*OF refl*] *sfilter-const-TT*])
next
  case *False* **then show** *?thesis*
    by (*clarsimp intro*!: *trans*[*OF sfilter-cong*[*OF refl*] *sfilter-const-TT*])
qed

lemma *endswith-eq-sdrop*: *endswith·pat·xs = eq·pat·(sdrop·(slength·xs − slength·pat)·xs)*
proof(*cases pat = ⊥ xs = ⊥ rule*: *bool.exhaust*[*case-product bool.exhaust*])
  case *False-False* **then show** *?thesis*
    by (*cases endswith·pat·xs*;
      simp only: *endswith.simps cfcomp2 stails-sdrop′*;
      force simp: *zero-Integer-def one-Integer-def elim*: *slengthE*)
qed *simp-all*

lemma *endswith-def2*: — Bird p127
  shows *endswith·pat·xs = eq·pat·(shead·(sfilter·(Λ x. prefix·x·pat)·(stails·xs)))* (**is** *?lhs = ?rhs*)
proof(*cases pat = ⊥ xs = ⊥ rule*: *bool.exhaust*[*case-product bool.exhaust*])
  case *False-False*
  from *False-False* **obtain** *patl xsl* **where** ∗: *slength·xs = MkI·xsl 0 ≤ xsl slength·pat = MkI·patl 0 ≤ patl*
    by (*meson Integer.exhaust slength-bottom-iff slength-ge-0*)
  let *?patl-xsl = if patl ≤ xsl then xsl − patl else 0*
  have *?rhs = eq·pat·(shead·(sfilter·(Λ x. le·(slength·x)·(slength·pat) andalso prefix·x·pat)·(stails·xs)))*
    by (*subst prefix-slength-strengthen*) *simp*
  also have . . . *= eq·pat·(shead·(sfilter·(Λ x. prefix·x·pat)·(sfilter·(Λ x. le·(slength·x)·(slength·pat))·(stails·xs))))*
    by (*simp add*: *sfilter-sfilter′*)
  also have . . . *= eq·pat·(shead·(smap·(Λ k. sdrop·k·xs)·(sfilter·(Λ k. prefix·(sdrop·k·xs)·pat)·(sfilter·(Λ k. le·(slength·(sdro*
    using ‹*slength·xs = MkI·xsl*› ‹*slength·pat = MkI·patl*›
    by (*simp add*: *stails-sdrop′ sfilter-smap′ cfcomp1 zero-Integer-def*)
  also have . . . *= eq·pat·(shead·(smap·(Λ k. sdrop·k·xs)·(sfilter·(Λ k. prefix·(sdrop·k·xs)·pat)·(sfilter·(Λ k. le·(MkI·xsl − k)·(MkI·patl))·(supto·(MkI·0)·(MkI·xsl))))))*
    using ‹*slength·xs = MkI·xsl*›
    by (*subst* (*2*) *sfilter-cong*[**where** *p′=Λ x. le·(MkI·xsl − x)·(MkI·patl)*]; *fastforce simp*: *zero-Integer-def*)
  also have . . . *= If prefix·(sdrop·(MkI·?patl-xsl)·xs)·pat*
      then eq·pat·(sdrop·(MkI·?patl-xsl)·xs)
      else eq·pat·(shead·(smap·(Λ k. sdrop·k·xs)·(sfilter·(Λ x. prefix·(sdrop·x·xs)·pat)·(supto·(MkI·(?patl-xsl + 1))·(MkI·xsl)))))
    using *False-False* ‹*0 ≤ xsl*› ‹*0 ≤ patl*›
    by (*subst sfilter-supto*) (*auto simp*: *If-distr one-Integer-def*)
  also have . . . *= ?lhs* (**is** *If ?c then - else ?else = -*)
  proof(*cases ?c*)
    case *TT* **with** ‹*slength·xs = MkI·xsl*› ‹*slength·pat = MkI·patl*›
    show *?thesis* **by** (*simp add*: *endswith-eq-sdrop sdrop-neg zero-Integer-def*)

next
case *FF* — Recursive case: the lists generated by *supto* are too short
have *?else* = *shead·(smap·(Λ x. eq·pat·(sdrop·x·xs))·(sfilter·(Λ x. prefix·(sdrop·x·xs)·pat)·(supto·(MkI·(?patl-xsl + 1))·(MkI·xsl))))*
using *FF* **by** (*subst shead-smap-distr*[**where** *f=eq·pat*, *symmetric*]) (*auto simp*: *cfcomp1*)
also have ... = *shead·(smap·(Λ x. seq·x·FF)·(sfilter·(Λ x. prefix·(sdrop·x·xs)·pat)·(supto·(MkI·(?patl-xsl + 1))·(MkI·xsl))))*
using *False-False* ∗ **by** (*subst smap-cong*[*OF refl*, **where** *f′*=Λ *x. seq·x·FF*]) (*auto simp*: *zero-Integer-def split*: *if-splits*)
also from ∗ *FF* **have** ... = *?lhs*
apply (*auto 0 0 simp*: *shead-smap-distr seq-conv-if endswith-eq-sdrop zero-Integer-def dest*!: *prefix-FF-not-snilD*)
apply (*metis* (*full-types*) *le-MkI-MkI linorder-not-less order-refl prefix-FF-not-snilD sdrop-all zless-imp-add1-zle*)
using *FF* **apply** *fastforce*
apply (*metis add.left-neutral le-MkI-MkI linorder-not-less order-refl prefix-FF-not-snilD sdrop-0*(*1*) *sdrop-all zero-Integer-def zless-imp-add1-zle*)
done
finally show *?thesis* **using** *FF* **by** *simp*
qed (*simp add*: *False-False*)
finally show *?thesis* **..**
qed *simp-all*

Bird then generalizes *sfilter·(Λ x. prefix·x·pat) oo stails* to *split*, where "*split·pat·xs* splits *pat* into two lists *us* and *vs* so that *us* :@ *vs* = *pat* and *us* is the longest suffix of *xs* that is a prefix of *pat*."

fixrec *split* :: [:′*a*::*Eq-def*:] → [:′*a*:] → [:′*a*:] × [:′*a*:] **where** — Bird p128
[*simp del*]: *split·pat·xs* = *If prefix·xs·pat then* (*xs*, *sdrop·(slength·xs)·pat*) *else split·pat·(stail·xs)*

lemma *split-strict*[*simp*]:
shows *split·*⊥ = ⊥
and *split·pat·*⊥ = ⊥
by *fixrec-simp*+

lemma *split-bottom-iff*[*simp*]: (*split·pat·xs* = ⊥) ⟷ (*pat* = ⊥ ∨ *xs* = ⊥)
by (*cases pat* = ⊥; *clarsimp*) (*induct xs*; *subst split.unfold*; *simp*)

lemma *split-snil*[*simp*]:
assumes *pat* ≠ ⊥
shows *split·pat·*[::] = ([::], *pat*)
using *assms* **by** *fixrec-simp*

lemma *split-pattern*: — Bird p128, observation
assumes *xs* ≠ ⊥
assumes *split·pat·xs* = (*us*, *vs*)
shows *us* :@ *vs* = *pat*
using *assms*
proof(*cases pat* = ⊥, *simp*, *induct xs arbitrary*: *us vs*)
case *snil* **then show** *?case* **by** (*subst* (*asm*) *split.unfold*) *simp*
next
case (*scons x xs*) **then show** *?case*
by (*subst* (*asm*) (*3*) *split.unfold*)
(*fastforce dest*: *prefix-sdrop-slength simp*: *If2-def*[*symmetric*] *split*: *If2-splits*)
qed *simp*

lemma *endswith-split*: — Bird p128, after defining *split*
shows *endswith·pat* = *snull oo csnd oo split·pat*
proof(*rule cfun-eqI*)
fix *xs* **show** *endswith·pat·xs* = (*snull oo csnd oo split·pat*)*·xs*
proof(*cases pat* = ⊥, *simp*, *induct xs*)
case (*scons x xs*) **then show** *?case*

unfolding *endswith-def2*
by (*subst split.unfold*)
(*fastforce dest*: *prefix-sdrop-prefix-eq simp*: *If2-def*[*symmetric*] *If-distr snull-eq-snil split*: *If2-splits*)
qed (*simp-all add*: *snull-eq-snil endswith.simps*)
qed

lemma *split-length-lt*:
assumes $pat \neq \bot$
assumes $xs \neq \bot$
shows *lt*·(*slength*·(*prod.fst* (*split*·*pat*·*xs*)))·(*slength*·*xs* + *1*) = *TT*
using *assms*
proof(*induct xs*)
case (*scons x xs*) **then show** *?case*
by (*subst split.unfold*)
(*auto simp*: *If2-def*[*symmetric*] *one-Integer-def split*: *If2-splits elim*!: *slengthE elim*: *lt-trans*)
qed *simp-all*

The predicate *p* required by *Bird-strategy* is therefore *snull oo csnd*. It remains to find *op* and *z* such that:

- *split*·*pat*·[::] = *z*
- *split*·*pat*·(*xs* :@ [:*x*:]) = *op*·(*split*·*pat*·*xs*)·*x*

so that *split* = *sfoldl*·*op*·*z*.
We obtain *z* = ([::], *pat*) directly from the definition of *split*.
Bird derives *op* on the basis of this crucial observation:

lemma *split-snoc*: — Bird p128
shows *split*·*pat*·(*xs* :@ [:*x*:]) = *split*·*pat*·(*cfst*·(*split*·*pat*·*xs*) :@ [:*x*:])
proof(*cases pat* = ⊥, *simp*, *cases x* = ⊥, *simp*, *induct xs*)
case (*scons x xs*) **then show** *?case*
apply −
apply (*subst* (*1 3*) *split.unfold*)
apply (*clarsimp simp*: *If2-def*[*symmetric*] *split*: *If2-splits*; *intro conjI impI*)
apply (*subst split.unfold*; *fastforce*)
apply (*subst split.unfold*; *fastforce*)
apply (*simp add*: *append-prefixD*)
done
qed *simp-all*

fixrec — Bird p129
op :: [:′*a*::*Eq-def*:] → [:′*a*:] × [:′*a*:] → ′*a* → [:′*a*:] × [:′*a*:]
where
[*simp del*]:
op·*pat*·(*us*, *vs*)·*x* =
(*If prefix*·[:*x*:]·*vs then* (*us* :@ [:*x*:], *stail*·*vs*)
else If snull·*us then* ([::], *pat*)
else op·*pat*·(*split*·*pat*·(*stail*·*us*))·*x*)

lemma *op-strict*[*simp*]:
op·*pat*·⊥ = ⊥
op·*pat*·(*us*, ⊥) = ⊥
op·*pat*·*usvs*·⊥ = ⊥
by *fixrec-simp*+

Bird demonstrates that *op* is partially correct wrt *split*, i.e., *op*·*pat*·(*split*·*pat*·*xs*)·*x* ⊑ *split*·*pat*·(*xs* :@ [:*x*:]). For total correctness we essentially prove that *op* terminates on well-defined arguments with an inductive argument.

lemma *op-induct*[*case-names step*]:

```
  fixes usvs:: [:'a:] × 'b
  assumes step: ⋀usvs. (⋀usvs'. lt·(slength·(cfst·usvs'))·(slength·(cfst·usvs)) = TT ⟹ P usvs') ⟹ P usvs
  shows P usvs
proof(induct usvs rule: wf-induct)
  let ?r = { (usvs', usvs) |(usvs :: [:'a:] × 'b) (usvs' :: [:'a:] × 'b). lt·(slength·(cfst·usvs'))·(slength·(cfst·usvs)) =
TT }
  show wf ?r
  proof(rule wf-subset[OF wf-inv-image[where f=λ(x, -). slength·x, OF wf-subset[OF wf-Integer-ge-less-than[where
d=0]]]])
    let ?rslen = { (slength·us', slength·us) |(us :: [:'a:]) (us' :: [:'a:]). lt·(slength·us')·(slength·us) = TT }
    show ?rslen ⊆ Integer-ge-less-than 0
      apply (clarsimp simp: Integer-ge-less-than-def zero-Integer-def)
      apply (metis Integer.exhaust dist-eq-tr(4) dist-eq-tr(6) lt-Integer-bottom-iff lt-MkI-MkI slength-ge-0)
      done
    show ?r ⊆ inv-image ?rslen (λ(x, -). slength·x) by (auto 0 3)
  qed
  fix usvs
  assume ∀ usvs'. (usvs', usvs) ∈ ?r ⟶ P usvs'
  then show P usvs
    by − (rule step; clarsimp; metis eq-fst-iff)
qed

lemma op-induct'[case-names step]:
  assumes step: ⋀us. (⋀us'. lt·(slength·us')·(slength·us) = TT ⟹ P us') ⟹ P us
  shows P us
by (rule op-induct[where P=P ∘ prod.fst and usvs=(us, vs) for vs::unit, simplified])
   (fastforce intro: step)

lemma split-snoc-op:
  split·pat·(xs :@ [:x:]) = op·pat·(split·pat·xs)·x
proof(induct split·pat·xs arbitrary: xs rule: op-induct)
  case (step xs) show ?case
  proof(cases pat = ⊥ xs = ⊥ x = ⊥ rule: bool.exhaust[case-product bool.exhaust bool.exhaust])
    case False-False-False
    obtain us vs where ∗: split·pat·xs = (us, vs) by fastforce
    from False-False-False ∗ have ∗∗: prefix·(us :@ [:x:])·pat = prefix·[:x:]·vs
      using split-pattern same-prefix-prefix sappend-bottom-iff by blast
    from False-False-False ∗ ∗∗
    have ∗∗∗: sdrop·(slength·(us :@ [:x:]))·pat = stail·vs if prefix·(us :@ [:x:])·pat = TT
      using sdrop-sappend-same[where xs=us :@ [:x:]] that
      by (cases vs; clarsimp) (fastforce dest!: split-pattern)
    from False-False-False ∗ ∗∗ ∗∗∗ show ?thesis
      apply −
      apply (subst split-snoc)
      apply (subst split.unfold)
      apply (subst op.unfold)
      apply (fastforce simp: If2-def[symmetric] snull-FF-conv split: If2-splits intro: step split-length-lt)
      done
  qed simp-all
qed

lemma split-sfoldl-op:
  assumes pat ≠ ⊥
  shows sfoldl·(op·pat)·([::], pat) = split·pat (is ?lhs = ?rhs)
proof −
  have ?lhs·xs = ?rhs·xs for xs
    using assms by (induct xs rule: srev-induct) (simp-all add: split-snoc-op)
  then show ?thesis by (simp add: cfun-eq-iff)
```

qed

lemma *matches-op*:
  shows *matches·pat* = *smap·cfst oo sfilter·*(*snull oo csnd oo csnd*)
      oo sscanl·(Λ (*n*, *usvs*) *x*. (*n* + *1*, *op·pat·usvs·x*))·(*0*, ([::], *pat*)) (**is** *?lhs* = *?rhs*)
proof(*cases pat* = ⊥)
  case *True*
  then have *?lhs·xs* = *?rhs·xs* **for** *xs* **by** (*cases xs*; *clarsimp*)
  then show *?thesis* **by** (*simp add*: *cfun-eq-iff*)
next
  case *False* **then show** *?thesis*
    apply (*subst* (*2*) *oo-assoc*)
    apply (*rule Bird-strategy*)
    apply (*simp-all add*: *endswith-split split-sfoldl-op oo-assoc*)
    done
qed

Using *split-sfoldl-op* we can rewrite *op* into a more perspicuous form that exhibits how KMP handles the failure of the text to continue matching the pattern:

fixrec
  op′ :: [:′*a*::*Eq-def*:] → [:′*a*:] × [:′*a*:] → ′*a* → [:′*a*:] × [:′*a*:]
where
[*simp del*]:
  op′·pat·(*us*, *vs*)·*x* =
    (*If prefix·*[:*x*:]·*vs then* (*us* :@ [:*x*:], *stail·vs*) — continue matching
     else If snull·us then ([::], *pat*) — fail at the start of the pattern: discard *x*
     else sfoldl·(*op′·pat*)·([::], *pat*)·(*stail·us* :@ [:*x*:]) — fail later: discard *shead·us* and determine where to restart
)

Intuitively if *x* continues the pattern match then we extend the *split* of *pat* recorded in *us* and *vs*. Otherwise we need to find a prefix of *pat* to continue matching with. If we have yet to make any progress (i.e., *us* = [::]) we restart with the entire *pat* (aka *z*) and discard *x*. Otherwise, because a match cannot begin with *us* :@ [:*x*:], we *split pat* (aka *z*) by iterating *op′* over *stail·us* :@ [:*x*:]. The remainder of the development is about memoising this last computation.

This is not yet the full KMP algorithm as it lacks what we call the 'K' optimisation, which we add in §4.2. Note that a termination proof for *op′* in HOL is tricky due to its use of higher-order nested recursion via *sfoldl*.

lemma *op′-strict*[*simp*]:
  op′·pat·⊥ = ⊥
  op′·pat·(*us*, ⊥) = ⊥
  op′·pat·usvs·⊥ = ⊥
by *fixrec-simp*+

lemma *sfoldl-op′-strict*[*simp*]:
  op′·pat·(*sfoldl·*(*op′·pat*)·(*us*, ⊥)·*xs*)·*x* = ⊥
by (*induct xs arbitrary*: *x rule*: *srev-induct*) *simp-all*

lemma *op′-op*:
  shows *op′·pat·usvs·x* = *op·pat·usvs·x*
proof(*cases pat* = ⊥ *x* = ⊥ *rule*: *bool.exhaust*[*case-product bool.exhaust*])
  case *True-False* **then show** *?thesis*
    apply (*subst op′.unfold*)
    apply (*subst op.unfold*)
    apply *simp*
    done
next
  case *False-False* **then show** *?thesis*
  proof(*induct usvs arbitrary*: *x rule*: *op-induct*)
    case (*step usvs x*)

have ∗: *sfoldl*·(*op*′·*pat*)·([::], *pat*)·*xs* = *sfoldl*·(*op*·*pat*)·([::], *pat*)·*xs*
if *lt*·(*slength*·*xs*)·(*slength*·(*cfst*·*usvs*)) = *TT* **for** *xs*
using *that*
proof(*induct xs rule*: *srev-induct*)
case (*ssnoc x*′ *xs*′)
from *ssnoc*(*1*,*2*,*4*) **have** *lt*·(*slength*·*xs*′)·(*slength*·(*cfst*·*usvs*)) = *TT*
using *lt-slength-0*(*2*) *lt-trans* **by** *auto*
moreover
from *step*(*2*) *ssnoc*(*1*,*2*,*4*) **have** *lt*·(*slength*·(*cfst*·(*split*·*pat*·*xs*′)))·(*slength*·(*cfst*·*usvs*)) = *TT*
using *lt-trans split-length-lt* **by** (*auto 10 0*)
ultimately show *?case* **by** (*simp add*: *ssnoc.hyps split-sfoldl-op split-snoc-op step*)
qed *simp-all*
from *step.prems* **show** *?case*
apply (*subst op*′.*unfold*)
apply (*subst op.unfold*)
apply (*clarsimp simp*: *If2-def*[*symmetric*] *snull-FF-conv split-sfoldl-op*[*symmetric*] ∗ *split*: *If2-splits*)
apply (*clarsimp simp*: *split-sfoldl-op step split-length-lt*)
done
qed
qed *simp-all*

4.2 Step 2: Data refinement and the 'K' optimisation

Bird memoises the restart computation in *op*′ in two steps. The first reifies the control structure of *op*′ into a non-wellfounded tree, which we discuss here. The second increases the sharing in this tree; see §4.6.

Briefly, we cache the *sfoldl*·(*op*′·*pat*)·([::], *pat*)·(*stail*·*us* :@ [:*x*:]) computation in *op*′ by finding a "representation" type ′*t* for the "abstract" type [:′*a*:] × [:′*a*:], a pair of functions *rep*, *abs* where *abs oo rep* = *ID*, and then finding a derived form of *op*′ that works on ′*t* rather than [:′*a*:] × [:′*a*:]. We also take the opportunity to add the 'K' optimisation in the form of the *next* function.

As such steps are essentially *deus ex machina*, we try to provide some intuition after showing the new definitions.

domain ′*a tree* — Bird p130
= *Null*
| *Node* (*label* :: ′*a*) (**lazy** *left* :: ′*a tree*) (**lazy** *right* :: ′*a tree*) — Strict in the label ′*a*

fixrec *next* :: [:′*a*::*Eq-def*:] → ([:′*a*:] × [:′*a*:]) *tree* → ([:′*a*:] × [:′*a*:]) *tree* **where**
next·[::]·*t* = *t*
| ⟦*x* ≠ ⊥; *xs* ≠ ⊥⟧ ⟹
next·(*x* :# *xs*)·*Null* = *Null*
| ⟦*x* ≠ ⊥; *xs* ≠ ⊥⟧ ⟹
next·(*x* :# *xs*)·(*Node*·(*us*, [::])·*l*·*r*) = *Node*·(*us*, [::])·*l*·*r*
| ⟦*v* ≠ ⊥; *vs* ≠ ⊥; *x* ≠ ⊥; *xs* ≠ ⊥⟧ ⟹
next·(*x* :# *xs*)·(*Node*·(*us*, *v* :# *vs*)·*l*·*r*) = *If eq*·*x*·*v then l else Node*·(*us*, *v* :# *vs*)·*l*·*r*

fixrec — Bird p131 "an even simpler form", with the 'K' optimisation
root2 :: [:′*a*::*Eq-def*:] → ([:′*a*:] × [:′*a*:]) *tree*
and *op2* :: [:′*a*:] → ([:′*a*:] × [:′*a*:]) *tree* → ′*a* → ([:′*a*:] × [:′*a*:]) *tree*
and *rep2* :: [:′*a*:] → [:′*a*:] × [:′*a*:] → ([:′*a*:] × [:′*a*:]) *tree*
and *left2* :: [:′*a*:] → [:′*a*:] × [:′*a*:] → ([:′*a*:] × [:′*a*:]) *tree*
and *right2* :: [:′*a*:] → [:′*a*:] × [:′*a*:] → ([:′*a*:] × [:′*a*:]) *tree*
where
[*simp del*]:
root2·*pat* = *rep2*·*pat*·([::], *pat*)
| *op2*·*pat*·*Null*·*x* = *root2*·*pat*
| *usvs* ≠ ⊥ ⟹
op2·*pat*·(*Node*·*usvs*·*l*·*r*)·*x* = *If prefix*·[:*x*:]·(*csnd*·*usvs*) *then r else op2*·*pat*·*l*·*x*
| [*simp del*]:
rep2·*pat*·*usvs* = *Node*·*usvs*·(*left2*·*pat*·*usvs*)·(*right2*·*pat*·*usvs*)

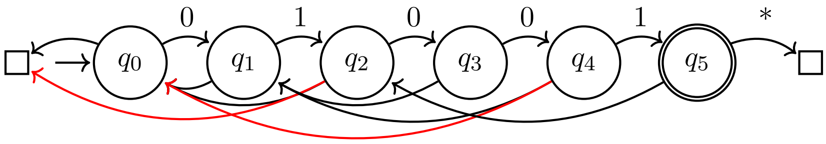


Figure 2: An example from Crochemore and Rytter (2002, §2.1). The MP tree for the pattern 01001 is drawn in black: right transitions are labelled with a symbol, whereas left transitions are unlabelled. The two 'K'-optimised left transitions are shown in red. The boxes denote *Null*. The root node is q_0.

| *left2·pat·*([::], *vs*) = *next·vs·Null*
| ⟦*u* ≠ ⊥; *us* ≠ ⊥⟧ ⟹
  left2·pat·(*u* :# *us*, *vs*) = *next·vs·*(*sfoldl·*(*op2·pat*)·(*root2·pat*)·*us*) — Note the use of *op2* and *next*.
| *right2·pat·*(*us*, [::]) = *Null* — Unreachable
| ⟦*v* ≠ ⊥; *vs* ≠ ⊥⟧ ⟹
  right2·pat·(*us*, *v* :# *vs*) = *rep2·pat·*(*us* :@ [:*v*:], *vs*)

fixrec *abs2* :: ([:'*a*:] × [:'*a*:]) *tree* → [:'*a*:] × [:'*a*:] **where**
 usvs ≠ ⊥ ⟹ *abs2·*(*Node·usvs·l·r*) = *usvs*

fixrec *matches2* :: [:'*a*::*Eq-def*:] → [:'*a*:] → [:*Integer*:] **where**
[*simp del*]: *matches2·pat* = *smap·cfst oo sfilter·*(*snull oo csnd oo abs2 oo csnd*)
  oo sscanl·(Λ (*n*, *x*) *y*. (*n* + *1*, *op2·pat·x·y*))·(*0*, *root2·pat*)

This tree can be interpreted as a sort of automaton[1], where *op2* goes *right* if the pattern continues with the next element of the text, and *left* otherwise, to determine how much of a prefix of the pattern could still be in play. Figure 2 visualises such an automaton for the pattern 01001, used by Crochemore and Rytter (2002, §2.1) to illustrate the difference between Morris-Pratt (MP) and Knuth-Morris-Pratt (KMP) preprocessing as we discuss below. Note that these are not the classical Mealy machines that correspond to regular expressions, where all outgoing transitions are labelled with symbols.

The following lemma shows how our sample automaton is encoded as a non-wellfounded tree.

lemma *concrete-tree-KMP*:
 shows *root2·*[:*0*::*Integer*, *1*, *0*, *0*, *1*:]
  = (μ *q0*. *Node·*([::], [:*0*, *1*, *0*, *0*, *1*:])
    ·*Null*
    ·(μ *q1*. *Node·*([:*0*:], [:*1*, *0*, *0*, *1*:])
      ·*q0*
      ·(μ *q2*. *Node·*([:*0*,*1*:], [:*0*, *0*, *1*:])
        ·*Null* — K optimisation: MP *q0*
        ·(*Node·*([:*0*,*1*,*0*:], [:*0*, *1*:])
          ·*q1*
          ·(*Node·*([:*0*,*1*,*0*,*0*:], [:*1*:])
            ·*q0* — K optimisation: MP *q1*
            ·(*Node·*([:*0*,*1*,*0*,*0*,*1*:], [::])·*q2*·*Null*))))))
(**is** *?lhs* = *fix·?F*)

The sharing that we expect from a lazy (call-by-need) evaluator is here implied by the use of nested fixed points.

The KMP preprocessor is expressed by the *left2* function, where *op2* is used to match the pattern against itself; the use of *op2* in *matches2* ("the driver") is responsible for matching the (preprocessed) pattern against the text. This formally cashes in an observation by van der Woude (1989, §5), that these two algorithms are essentially the same, which has eluded other presentations[2].

Bird uses *Null* on a left path to signal to the driver that it should discard the current element of the text and restart matching from the beginning of the pattern (i.e, *root2*). This is a step towards the removal of *us* in §4.8.

Note that the *Null* at the end of the rightmost path is unreachable: the rightmost *Node* has *vs* = [::] and therefore *op2* always takes the left branch.

[1] Bird (2012, §3.1) suggests it can be thought of as a doubly-linked list, following Takeichi and Akama (1991).)

[2] For instance, contrast our shared use of *op2* with the separated `match` and `rematch` functions of Ager et al. (2006, Figure 1).

The 'K' optimisation is perhaps best understood by example. Consider the automaton in Figure 2, and a text beginning with 011. Using the MP (black) transitions we take the path $\rightarrow q_0 \overset{0}{\rightarrow} q_1 \overset{1}{\rightarrow} \overbrace{q_2 \rightarrow q_0 \rightarrow \Box}$. Now, due to the failure of the comparison of the current element of the text (1) at q_2, we can predict that the (identical) comparison at node q_0 will fail as well, and therefore have q_2 left-branch directly to $\Box$. This saves a comparison in the driver at the cost of another in the preprocessor (in *next*). These optimisations are the red arrows in the diagram, and can in general save an arbitrary number of driver comparisons; consider the pattern 1^n for instance.

More formally, *next* ensures that the heads of the suffixes of the pattern (*vs*) on consecutive labels on left paths are distinct; see below for a proof of this fact in our setting, and Gusfield (1997, §3.3.4) for a classical account. Unlike Bird's suggestion (p134), our *next* function is not recursive.

We note in passing that while MP only allows *Null* on the left of the root node, *Null* can be on the left of any KMP node except for the rightmost (i.e., the one that signals a complete pattern match) where no optimisation is possible.

We proceed with the formalities of the data refinement.

schematic-goal *root2-op2-rep2-left2-right2-def*: — Obtain the definition of these functions as a single fixed point
  (*root2* :: [:'*a*::*Eq-def*:] → ([:'*a*:] × [:'*a*:]) *tree*
  , *op2* :: [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree* → '*a* → ([:'*a*:] × [:'*a*:]) *tree*
  , *rep2* :: [:'*a*:] → [:'*a*:] × [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree*
  , *left2* :: [:'*a*:] → [:'*a*:] × [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree*
  , *right2* :: [:'*a*:] → [:'*a*:] × [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree*)
  = *fix*·?*F*
unfolding *op2-def root2-def rep2-def left2-def right2-def* **by** *simp*

lemma *abs2-strict*[*simp*]:
  abs2·⊥ = ⊥
  abs2·*Null* = ⊥
by *fixrec-simp*+

lemma *next-strict*[*simp*]:
  next·⊥ = ⊥
  next·*xs*·⊥ = ⊥
  next·(*x* :# *xs*)·(*Node*·(*us*, ⊥)·*l*·*r*) = ⊥
  apply *fixrec-simp*
 apply (*cases xs*; *fixrec-simp*; *simp*)
apply (*cases x* = ⊥; *cases xs* = ⊥; *cases us* = ⊥; *fixrec-simp*)
done

lemma *next-Null*[*simp*]:
  assumes *xs* ≠ ⊥
shows *next*·*xs*·*Null* = *Null*
using *assms* **by** (*cases xs*) *simp-all*

lemma *next-snil*[*simp*]:
  assumes *xs* ≠ ⊥
  shows *next*·*xs*·(*Node*·(*us*, [::])·*l*·*r*) = *Node*·(*us*, [::])·*l*·*r*
using *assms* **by** (*cases xs*) *simp-all*

lemma *op2-rep2-left2-right2-strict*[*simp*]:
  op2·*pat*·⊥ = ⊥
  op2·*pat*·(*Node*·(*us*, ⊥)·*l*·*r*) = ⊥
  op2·*pat*·(*Node*·*usvs*·*l*·*r*)·⊥ = ⊥
  rep2·*pat*·⊥ = ⊥
  left2·*pat*·(⊥, *vs*) = ⊥
  left2·*pat*·(*us*, ⊥) = ⊥
  right2·*pat*·(*us*, ⊥) = ⊥
    apply *fixrec-simp*
   apply (*cases us* = ⊥; *fixrec-simp*; *simp*)
  apply (*cases usvs* = ⊥; *fixrec-simp*; *simp*)

apply *fixrec-simp*
apply *fixrec-simp*
apply (*cases us*; *fixrec-simp*)
apply *fixrec-simp*
done

lemma *snd-abs-root2-bottom*[*simp*]: *prod.snd* (*abs2*·(*root2*·⊥)) = ⊥
by (*simp add*: *root2.unfold rep2.unfold*)

lemma *abs-rep2-ID′*[*simp*]: *abs2*·(*rep2*·*pat*·*usvs*) = *usvs*
by (*cases usvs* = ⊥; *subst rep2.unfold*; *clarsimp*)

lemma *abs-rep2-ID*: *abs2 oo rep2*·*pat* = *ID*
by (*clarsimp simp*: *cfun-eq-iff*)

lemma *rep2-snoc-right2*: — Bird p131
assumes *prefix*·[:*x*:]·*vs* = *TT*
shows *rep2*·*pat*·(*us* :@ [:*x*:], *stail*·*vs*) = *right2*·*pat*·(*us*, *vs*)
using *assms* **by** (*cases x* = ⊥; *cases vs*; *clarsimp*)

lemma *not-prefix-op2-next*:
assumes *prefix*·[:*x*:]·*xs* = *FF*
shows *op2*·*pat*·(*next*·*xs*·(*rep2*·*pat*·*usvs*))·*x* = *op2*·*pat*·(*rep2*·*pat*·*usvs*)·*x*
proof −
obtain *us vs* **where** *usvs* = (*us*, *vs*) **by** *force*
with *assms* **show** *?thesis*
by (*cases xs*; *cases us*; *clarsimp*; *cases vs*;
clarsimp simp: *rep2.simps prefix-singleton-FF If2-def*[*symmetric*] *split*: *If2-splits*)
qed

Bird's appeal to *foldl-fusion* (p130) is too weak to justify this data refinement as his condition (iii) requires the worker functions to coincide on all representation values. Concretely he asks that:

$$rep2\cdot pat\cdot(op\cdot pat\cdot(abs2\cdot t)\cdot x) = op2\cdot pat\cdot t\cdot x \text{ — Bird (17.2)}$$

where *t* is an arbitrary tree. This does not hold for junk representations such as:

$$t = Node\cdot(pat, [::])\cdot Null\cdot Null$$

Using worker/wrapper fusion (Gammie 2011; Gill and Hutton 2009) specialised to *sscanl* (*sscanl-ww-fusion*) we only need to establish this identity for valid representations, i.e., when *t* lies under the image of *rep2*. In pictures, we show that this diagram commutes:

$$\begin{array}{ccc} usvs & \xrightarrow{\Lambda\ usvs.\ op\cdot pat\cdot usvs\cdot x} & usvs' \\ \downarrow{\scriptstyle rep2\cdot pat} & & \downarrow{\scriptstyle rep2\cdot pat} \\ t & \xrightarrow{\Lambda\ usvs.\ op2\cdot pat\cdot usvs\cdot x} & t' \end{array}$$

Clearly this result self-composes: after an initial *rep2*·*pat* step, we can repeatedly simulate *op* steps with *op2* steps.

lemma *op-op2-refinement*:
assumes *pat* ≠ ⊥
shows *rep2*·*pat*·(*op*·*pat*·*usvs*·*x*) = *op2*·*pat*·(*rep2*·*pat*·*usvs*)·*x*
proof(*cases x* = ⊥ *usvs* = ⊥ *rule*: *bool.exhaust*[*case-product bool.exhaust*])
case *False-False*
then have *x* ≠ ⊥ *usvs* ≠ ⊥ **by** *simp-all*
then show *?thesis*
proof(*induct usvs arbitrary*: *x rule*: *op-induct*)
case (*step usvs*)

obtain *us vs* **where** *usvs*: *usvs* = (*us*, *vs*) **by** *fastforce*
have ∗: *sfoldl·*(*op2·pat*)*·*(*root2·pat*)*·xs* = *rep2·pat·*(*split·pat·xs*) **if** *lt·*(*slength·xs*)*·*(*slength·us*) = *TT* **for** *xs*
using *that*
proof(*induct xs rule*: *srev-induct*)
case (*ssnoc x xs*)
from *ssnoc*(*1*,*2*,*4*) **have** *IH*: *sfoldl·*(*op2·pat*)*·*(*root2·pat*)*·xs* = *rep2·pat·*(*split·pat·xs*)
by − (*rule ssnoc*; *auto intro*: *lt-trans dest*: *lt-slength-0*)
obtain *us′ vs′* **where** *us′vs′*: *split·pat·xs* = (*us′*, *vs′*) **by** *fastforce*
from ‹*pat* ≠ ⊥› *ssnoc*(*1*,*2*,*4*) *usvs* **show** *?case*
apply (*clarsimp simp*: *split-sfoldl-op*[*symmetric*] *IH*)
apply (*rule step*(*1*)[*simplified abs-rep2-ID′*, *simplified*, *symmetric*])
using *lt-trans split-length-lt split-sfoldl-op* **apply** *fastforce*+
done
qed (*fastforce simp*: ‹*pat* ≠ ⊥› *root2.unfold*)+
have ∗∗: *If snull·us then rep2·pat·*([::], *pat*) *else rep2·pat·*(*op·pat·*(*split·pat·*(*stail·us*))*·x*)
= *op2·pat·*(*left2·pat·*(*us*, *vs*))*·x* **if** *prefix·*[:*x*:]*·vs* = *FF*
proof(*cases us*)
case *snil* **with** *that* **show** *?thesis*
by *simp* (*metis next-Null op2.simps*(*1*) *prefix.simps*(*1*) *prefix-FF-not-snilD root2.simps*)
next
case (*scons u′ us′*)
from ‹*pat* ≠ ⊥› *scons* **have** *lt·*(*slength·*(*cfst·*(*split·pat·us′*)))*·*(*slength·us*) = *TT*
using *split-length-lt* **by** *fastforce*
from ‹*pat* ≠ ⊥› ‹*x* ≠ ⊥› *usvs that scons this* **show** *?thesis*
by (*clarsimp simp*: ∗ *step*(*1*)[*simplified abs-rep2-ID′*] *not-prefix-op2-next*)
qed *simp*
from ‹*usvs* ≠ ⊥› *usvs* **show** *?case*
apply (*subst* (*2*) *rep2.unfold*)
apply (*subst op2.unfold*)
apply (*subst op.unfold*)
apply (*clarsimp simp*: *If-distr rep2-snoc-right2* ∗∗ *cong*: *If-cong*)
done
qed
qed (*simp-all add*: *rep2.unfold*)

Therefore the result of this data refinement is extensionally equal to the specification:

lemma *data-refinement*:
shows *matches* = *matches2*
proof(*intro cfun-eqI*)
fix *pat xs* :: [:*′a*:] **show** *matches·pat·xs* = *matches2·pat·xs*
proof(*cases pat* = ⊥)
case *True* **then show** *?thesis* **by** (*cases xs*; *clarsimp simp*: *matches2.simps*)
next
case *False* **then show** *?thesis*
unfolding *matches2.simps*
apply (*subst matches-op*) — Continue with previous derivation.
apply (*subst sscanl-ww-fusion*[**where** *wrap*=*ID* ∗∗ *abs2* **and** *unwrap*=*ID* ∗∗ *rep2·pat* **and** *f′*=Λ (*n*, *x*) *y*. (*n* + *1*, *op2·pat·x·y*)])
apply (*simp add*: *abs-rep2-ID*)
apply (*simp add*: *op-op2-refinement*)
apply (*simp add*: *oo-assoc sfilter-smap root2.unfold*)
apply (*simp add*: *oo-assoc*[*symmetric*])
done
qed
qed

This computation can be thought of as a pair coroutines with a producer (*root2*/*rep2*) / consumer (*op2*) structure. It turns out that laziness is not essential (see §6), though it does depend on being able to traverse incompletely defined trees.

The key difficulty in defining this computation in HOL using present technology is that *op2* is neither terminating nor *friendly* in the terminology of Blanchette et al. (2017).

While this representation works for automata with this sort of structure, it is unclear how general it is; in particular it may not work so well if *left* branches can go forward as well as back. See also the commentary in Hinze and Jeuring (2001), who observe that sharing is easily lost, and so it is probably only useful in "closed" settings like the present one, unless the language is extended in unusual ways (Jeannin et al. 2017).

We conclude by proving that *rep2* produces trees that have the 'K' property, viz that labels on consecutive nodes on a left path do not start with the same symbol. This also establishes the productivity of *root2*. The pattern of proof used here – induction nested in coinduction – recurs in §4.6.

```
coinductive K :: ([:'a::Eq:] × [:'a:]) tree ⇒ bool where
  K Null
| ⟦ usvs ≠ ⊥; K l; K r;
     ⋀v vs. csnd·usvs = v :# vs ⟹ l = Null ∨ (∃ v' vs'. csnd·(label·l) = v' :# vs' ∧ eq·v·v' = FF)
  ⟧ ⟹ K (Node·usvs·l·r)

declare K.intros[intro!, simp]

lemma sfoldl-op2-root2-rep2-split:
  assumes pat ≠ ⊥
  shows sfoldl·(op2·pat)·(root2·pat)·xs = rep2·pat·(split·pat·xs)
proof(induct xs rule: srev-induct)
 case (ssnoc x xs) with ‹pat ≠ ⊥› ssnoc show ?case by (clarsimp simp: split-sfoldl-op[symmetric] op-op2-refinement)
qed (simp-all add: ‹pat ≠ ⊥› root2.unfold)

lemma K-rep2:
  assumes pat ≠ ⊥
  assumes us :@ vs = pat
  shows K (rep2·pat·(us, vs))
using assms
proof(coinduction arbitrary: us vs)
  case (K us vs) then show ?case
  proof(induct us arbitrary: vs rule: op-induct')
    case (step us)
    from step.prems have us ≠ ⊥ vs ≠ ⊥ by auto
    show ?case
    proof(cases us)
      case bottom with ‹us ≠ ⊥› show ?thesis by simp
    next
      case snil with step.prems show ?thesis by (cases vs; force simp: rep2.simps)
    next
      case (scons u' us')
      from ‹pat ≠ ⊥› scons ‹us ≠ ⊥› ‹vs ≠ ⊥›
      obtain usl vsl where splitl: split·pat·us' = (usl, vsl) usl ≠ ⊥ vsl ≠ ⊥ usl :@ vsl = pat
       by (metis (no-types, opaque-lifting) Rep-cfun-strict1 prod.collapse sappend-strict sappend-strict2 split-pattern)
      from scons obtain l r where r: rep2·pat·(us, vs) = Node·(us, vs)·l·r by (simp add: rep2.simps)
      moreover
      have (∃ us vs. l = rep2·pat·(us, vs) ∧ us :@ vs = pat) ∨ K l
      proof(cases vs)
        case snil with scons splitl r show ?thesis
          by (clarsimp simp: rep2.simps sfoldl-op2-root2-rep2-split)
      next
        case scons
        with ‹pat ≠ ⊥› ‹us = u' :# us'› ‹u' ≠ ⊥› ‹us' ≠ ⊥› ‹vs ≠ ⊥› r splitl show ?thesis
          apply (clarsimp simp: rep2.simps sfoldl-op2-root2-rep2-split)
          apply (cases vsl; cases usl; clarsimp simp: If2-def[symmetric] sfoldl-op2-root2-rep2-split split: If2-splits)
          apply (rename-tac ul' usl')
```

apply (*cut-tac us′=prod.fst* (*split·pat·usl′*) **and** *vs=prod.snd* (*split·pat·usl′*) **in** *step*(*1*); *clarsimp simp*: *split-pattern*)
apply (*metis fst-conv lt-trans slength.simps*(*2*) *split-length-lt step.prems*(*1*))
apply (*erule disjE*; *clarsimp simp*: *sfoldl-op2-root2-rep2-split*)
apply (*rename-tac b l r*)
apply (*case-tac b*; *clarsimp simp*: *rep2.simps*)
apply (*auto simp*: *If2-def*[*symmetric*] *rep2.simps dest*: *split-pattern*[*rotated*] *split*: *If2-splits*)
done
qed (*simp add*: ‹$vs \neq \bot$›)
moreover
from ‹*us* :@ *vs* = *pat*› ‹$us \neq \bot$› ‹$vs \neq \bot$› *r*
have ($\exists$ *usr vsr*. *r* = *rep2·pat·*(*usr*, *vsr*) $\wedge$ *usr* :@ *vsr* = *pat*) $\vee$ *K r*
by (*cases vs*; *clarsimp simp*: *rep2.simps*)
moreover
have *l* = *Null* $\vee$ ($\exists$ *v′ vs′*. *csnd·*(*label·l*) = *v′* :# *vs′* $\wedge$ *eq·v·v′* = *FF*) **if** *vs* = *v* :# *vs′* **for** *v vs′*
proof(*cases vsl*)
case *snil* **with** ‹*us* :@ *vs* = *pat*› ‹*us* = *u′* :# *us′*› *splitl* **show** *?thesis*
using *split-length-lt*[**where** *pat=pat* **and** *xs=us′*]
by (*force elim*: *slengthE simp*: *one-Integer-def split*: *if-splits*)
next
case *scons*
from *splitl* **have** *lt·*(*slength·usl*)*·*(*slength·us′* + *1*) = *TT*
by (*metis fst-conv fst-strict split-bottom-iff split-length-lt*)
with *scons* ‹$pat \neq \bot$› ‹*us* = *u′* :# *us′*› ‹$u' \neq \bot$› ‹$us' \neq \bot$› ‹$vs \neq \bot$› *r splitl* ‹*vs* = *v* :# *vs′*› **show** *?thesis*
using *step*(*1*)[*OF* - ‹$pat \neq \bot$›, **where** *us′=prod.fst* (*split·pat·us′*) **and** *vs=prod.snd* (*split·pat·us′*)]
by (*clarsimp simp*: *rep2.simps sfoldl-op2-root2-rep2-split If2-def*[*symmetric*] *split*: *If2-splits*)
qed (*simp add*: ‹$vsl \neq \bot$›)
moreover note ‹$pat \neq \bot$› ‹$us \neq \bot$› ‹$vs \neq \bot$›
ultimately show *?thesis* **by** *auto*
qed
qed
qed

theorem *K-root2*:
assumes $pat \neq \bot$
shows *K* (*root2·pat*)
using *assms* **unfolding** *root2.unfold* **by** (*simp add*: *K-rep2*)

The remaining steps are as follows:

- 3. introduce an accumulating parameter (*grep*).
- 4. inline *rep* and simplify.
- 5. simplify to Bird's "simpler forms."
- 6. memoise *left*.
- 7. simplify, unfold *prefix*.
- 8. discard *us*.
- 9. factor out *pat*.

4.3 Step 3: Introduce an accumulating parameter (grep)

Next we prepare for the second memoization step (§4.6) by introducing an accumulating parameter to *rep2* that supplies the value of the left subtree.

We retain *rep2* as a wrapper for now, and inline *right2* to speed up simplification.

fixrec — Bird p131 / p132
  root3 :: [:'a::*Eq-def*:] → ([:'a:] × [:'a:]) *tree*
and *op3* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
and *rep3* :: [:'a:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*
and *grep3* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*
where
 [*simp del*]:
 root3·*pat* = *rep3*·*pat*·([::], *pat*)
| *op3*·*pat*·*Null*·*x* = *root3*·*pat*
| *usvs* ≠ ⊥ ⟹
  op3·*pat*·(*Node*·*usvs*·*l*·*r*)·*x* = *If prefix*·[:*x*:]·(*csnd*·*usvs*) *then r else op3*·*pat*·*l*·*x*
| [*simp del*]: — Inline *left2*, factor out *next*.
 rep3·*pat*·*usvs* = *grep3*·*pat*·(*case cfst*·*usvs of* [::] ⇒ *Null* | *u* :# *us* ⇒ *sfoldl*·(*op3*·*pat*)·(*root3*·*pat*)·*us*)·*usvs*
| [*simp del*]: — *rep2* with *left2* abstracted, *right2* inlined.
 grep3·*pat*·*l*·*usvs* = *Node*·*usvs*·(*next*·(*csnd*·*usvs*)·*l*)·(*case csnd*·*usvs of*
   [::] ⇒ *Null*
  | *v* :# *vs* ⇒ *rep3*·*pat*·(*cfst*·*usvs* :@ [:*v*:], *vs*))

schematic-goal *root3-op3-rep3-grep3-def*:
 (*root3* :: [:'a::*Eq-def*:] → ([:'a:] × [:'a:]) *tree*
 , *op3* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
 , *rep3* :: [:'a:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*
 , *grep3* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*)
 = *fix*·?*F*
unfolding *root3-def op3-def rep3-def grep3-def* **by** *simp*

lemma *r3-2*:
 (Λ (*root*, *op*, *rep*, *grep*). (*root*, *op*, *rep*))·
 (*root3* :: [:'a::*Eq-def*:] → ([:'a:] × [:'a:]) *tree*
 , *op3* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
 , *rep3* :: [:'a:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*
 , *grep3* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*)
 = (Λ (*root*, *op*, *rep*, *left*, *right*). (*root*, *op*, *rep*))·
 (*root2* :: [:'a::*Eq-def*:] → ([:'a:] × [:'a:]) *tree*
 , *op2* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
 , *rep2* :: [:'a::*Eq-def*:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*
 , *left2* :: [:'a::*Eq-def*:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*
 , *right2* :: [:'a::*Eq-def*:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*)
unfolding *root2-op2-rep2-left2-right2-def root3-op3-rep3-grep3-def*
apply (*simp add*: *match-snil-match-scons-slist-case match-Null-match-Node-tree-case slist-case-distr tree-case-distr*)
apply (*simp add*: *fix-cprod fix-const*) — Very slow. Sensitive to tuple order due to the asymmetry of *fix-cprod*.
apply (*simp add*: *slist-case-distr*)
done

4.4 Step 4: Inline rep

We further simplify by inlining *rep3* into *root3* and *grep3*.

fixrec
  root4 :: [:'a::*Eq-def*:] → ([:'a:] × [:'a:]) *tree*
and *op4* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
and *grep4* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*
where
 [*simp del*]:
 root4·*pat* = *grep4*·*pat*·*Null*·([::], *pat*)
| *op4*·*pat*·*Null*·*x* = *root4*·*pat*
| *usvs* ≠ ⊥ ⟹
  op4·*pat*·(*Node*·*usvs*·*l*·*r*)·*x* = *If prefix*·[:*x*:]·(*csnd*·*usvs*) *then r else op4*·*pat*·*l*·*x*
| [*simp del*]:

grep4·pat·l·usvs = Node·usvs·(next·(csnd·usvs)·l)·(case csnd·usvs of
[::] ⇒ Null
| v :# vs ⇒ grep4·pat·(case cfst·usvs :@ [:v:] of
[::] ⇒ Null — unreachable
| u :# us ⇒ sfoldl·(op4·pat)·(root4·pat)·us)·(cfst·usvs :@ [:v:], vs))

schematic-goal *root4-op4-grep4-def*:
(root4 :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
, op4 :: [:'a:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
, grep4 :: [:'a:] → ([:'a:] × [:'a:]) tree → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree)
= fix·?F
unfolding *root4-def op4-def grep4-def* **by** *simp*

lemma *fix-syn4-permute*:
assumes cont (λ(X1, X2, X3, X4). F1 X1 X2 X3 X4)
assumes cont (λ(X1, X2, X3, X4). F2 X1 X2 X3 X4)
assumes cont (λ(X1, X2, X3, X4). F3 X1 X2 X3 X4)
assumes cont (λ(X1, X2, X3, X4). F4 X1 X2 X3 X4)
shows fix-syn (λ(X1, X2, X3, X4). (F1 X1 X2 X3 X4, F2 X1 X2 X3 X4, F3 X1 X2 X3 X4, F4 X1 X2 X3 X4))
= (λ(x1, x2, x4, x3). (x1, x2, x3, x4))
(fix-syn (λ(X1, X2, X4, X3). (F1 X1 X2 X3 X4, F2 X1 X2 X3 X4, F4 X1 X2 X3 X4, F3 X1 X2 X3 X4)))
by (induct rule: parallel-fix-ind) (use assms **in** ‹auto simp: prod-cont-iff›)

lemma *r4-3*:
(root4 :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
, op4 :: [:'a:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
, grep4 :: [:'a:] → ([:'a:] × [:'a:]) tree → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree)
= (Λ (root, op, rep, grep). (root, op, grep))·
(root3 :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
, op3 :: [:'a:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
, rep3 :: [:'a:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree
, grep3 :: [:'a:] → ([:'a:] × [:'a:]) tree → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree)
unfolding *root3-op3-rep3-grep3-def root4-op4-grep4-def*
apply (clarsimp simp: slist-case-distr match-Null-match-Node-tree-case tree-case-distr eta-cfun)
apply (subst fix-syn4-permute; clarsimp simp: fix-cprod fix-const) — Slow
done

4.5 Step 5: Simplify to Bird's "simpler forms"

The remainder of *left2* in *grep4* can be simplified by transforming the *case* scrutinee from *cfst·usvs* :@ [:*v*:] into *cfst·usvs*.

fixrec
root5 :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
and op5 :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
and grep5 :: [:'a:] → ([:'a:] × [:'a:]) tree → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree
where
[simp del]:
root5·pat = grep5·pat·Null·([::], pat)
| op5·pat·Null·x = root5·pat
| usvs ≠ ⊥ ⟹
op5·pat·(Node·usvs·l·r)·x = If prefix·[:x:]·(csnd·usvs) then r else op5·pat·l·x
| [simp del]:
grep5·pat·l·usvs = Node·usvs·(next·(csnd·usvs)·l)·(case csnd·usvs of
[::] ⇒ Null
| v :# vs ⇒ grep5·pat·(case cfst·usvs of — was cfst·usvs :@ [:v:]
[::] ⇒ root5·pat

```
          | u :# us ⇒ sfoldl·(op5·pat)·(root5·pat)·(us :@ [:v:]))·(cfst·usvs :@ [:v:], vs))

schematic-goal root5-op5-grep5-def:
  ( root5  :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
   , op5    :: [:'a:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
   , grep5  :: [:'a:] → ([:'a:] × [:'a:]) tree → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree )
   = fix·?F
unfolding root5-def op5-def grep5-def by simp

lemma op5-grep5-strict[simp]:
  op5·pat·⊥ = ⊥
  op5·pat·(Node·(us, ⊥)·l·r) = ⊥
  op5·pat·(Node·usvs·l·r)·⊥ = ⊥
  grep5·pat·l·⊥ = ⊥
   apply fixrec-simp
  apply (cases us = ⊥; fixrec-simp; simp)
 apply (cases usvs = ⊥; fixrec-simp; simp)
apply fixrec-simp
done

lemma r5-4:
  ( root5  :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
   , op5    :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
   , grep5  :: [:'a:] → ([:'a:] × [:'a:]) tree → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree )
 = ( root4  :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
   , op4    :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
   , grep4  :: [:'a:] → ([:'a:] × [:'a:]) tree → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree )
unfolding root4-op4-grep4-def root5-op5-grep5-def
by (clarsimp simp: slist-case-distr slist-case-snoc stail-sappend cong: slist-case-cong)
```

4.6 Step 6: Memoize left

The last substantial step is to memoise the computation of the left subtrees by tying the knot. Note this makes the computation of *us* in the tree redundant; we remove it in §4.8.

```
fixrec — Bird p132
    root6  :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
and op6    :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
and grep6  :: [:'a:] → ([:'a:] × [:'a:]) tree → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree
where
  [simp del]:
  root6·pat = grep6·pat·Null·([::], pat)
| op6·pat·Null·x = root6·pat
| usvs ≠ ⊥ ⟹
    op6·pat·(Node·usvs·l·r)·x = If prefix·[:x:]·(csnd·usvs) then r else op6·pat·l·x
| [simp del]:
  grep6·pat·l·usvs = Node·usvs·(next·(csnd·usvs)·l)·(case csnd·usvs of
          [::] ⇒ Null
        | v :# vs ⇒ grep6·pat·(op6·pat·l·v)·(cfst·usvs :@ [:v:], vs))

schematic-goal root6-op6-grep6-def:
  ( root6  :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
   , op6    :: [:'a:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
   , grep6  :: [:'a:] → ([:'a:] × [:'a:]) tree → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree )
   = fix·?F
unfolding root6-def op6-def grep6-def by simp

lemma op6-grep6-strict[simp]:
```

op6·pat·⊥ = ⊥
op6·pat·(Node·(us, ⊥)·l·r) = ⊥
op6·pat·(Node·usvs·l·r)·⊥ = ⊥
grep6·pat·l·⊥ = ⊥
apply *fixrec-simp*
apply (*cases us* = ⊥; *fixrec-simp*; *simp*)
apply (*cases usvs* = ⊥; *fixrec-simp*; *simp*)
apply *fixrec-simp*
done

Intuitively this step cashes in the fact that, in the context of *grep6·pat·l·usvs*, *sfoldl·(op6·pat)·(root6·pat)·us* is equal to *l*.

Connecting this step with the previous one is not simply a matter of equational reasoning; we can see this by observing that the right subtree of *grep5·pat·l·usvs* does not depend on *l* whereas that of *grep6·pat·l·usvs* does, and therefore these cannot be extensionally equal. Furthermore the computations of the corresponding *root*s do not proceed in lockstep: consider the computation of the left subtree.

For our purposes it is enough to show that the trees *root5* and *root6* are equal, from which it follows that *op6* = *op5* by induction on its tree argument. The equality is established by exhibiting a *tree bisimulation* (*tree-bisim*) that relates the corresponding "producer" *grep* functions. Such a relation *R* must satisfy:

- *R* ⊥ ⊥;
- *R Null Null*; and
- if *R* (*Node·x·l·r*) (*Node·x′·l′·r′*) then $x = x'$, *R l l′*, and *R r r′*.

The following pair of *left* functions define suitable left paths from the corresponding *root*s.

fixrec *left5* :: [:′*a*::*Eq-def*:] → [:′*a*:] → ([:′*a*:] × [:′*a*:]) *tree* **where**
left5·pat·[::] = Null
| ⟦*u* ≠ ⊥; *us* ≠ ⊥⟧ ⟹
left5·pat·(u :# us) = sfoldl·(op5·pat)·(root5·pat)·us

fixrec *left6* :: [:′*a*::*Eq-def*:] → [:′*a*:] → ([:′*a*:] × [:′*a*:]) *tree* **where**
left6·pat·[::] = Null
| ⟦*u* ≠ ⊥; *us* ≠ ⊥⟧ ⟹
left6·pat·(u :# us) = sfoldl·(op6·pat)·(root6·pat)·us

inductive — This relation is not inductive.
root-bisim :: [:′*a*::*Eq-def*:] ⇒ ([:′*a*:] × [:′*a*:]) *tree* ⇒ ([:′*a*:] × [:′*a*:]) *tree* ⇒ *bool*
for *pat* :: [:′*a*:]
where
bottom: *root-bisim pat* ⊥ ⊥
| *Null*: *root-bisim pat Null Null*
| *gl*: ⟦*pat* ≠ ⊥; *us* ≠ ⊥; *vs* ≠ ⊥⟧
⟹ *root-bisim pat* (*grep6·pat·(left6·pat·us)·(us, vs)*) (*grep5·pat·(left5·pat·us)·(us, vs)*)

declare *root-bisim.intros*[*intro*!, *simp*]

lemma *left6-left5-strict*[*simp*]:
left6·pat·⊥ = ⊥
left5·pat·⊥ = ⊥
by *fixrec-simp+*

lemma *op6-left6*: ⟦*us* ≠ ⊥; *v* ≠ ⊥⟧ ⟹ *op6·pat·(left6·pat·us)·v = left6·pat·(us :@ [:v:])*
by (*cases us*) *simp-all*

lemma *op5-left5*: ⟦*us* ≠ ⊥; *v* ≠ ⊥⟧ ⟹ *op5·pat·(left5·pat·us)·v = left5·pat·(us :@ [:v:])*
by (*cases us*) *simp-all*

lemma *root5-left5*: $v \neq \bot \Longrightarrow root5 \cdot pat = left5 \cdot pat \cdot [:v:]$
by *simp*

lemma *op5-sfoldl-left5*: $[\![us \neq \bot; u \neq \bot; v \neq \bot]\!] \Longrightarrow$
$op5 \cdot pat \cdot (sfoldl \cdot (op5 \cdot pat) \cdot (root5 \cdot pat) \cdot us) \cdot v = left5 \cdot pat \cdot (u :\# \ us :@ [:v:])$
by *simp*

lemma *root-bisim-root*:
  assumes $pat \neq \bot$
  shows *root-bisim pat* (*root6·pat*) (*root5·pat*)
unfolding *root6.unfold root5.unfold* **using** *assms*
by *simp* (*metis* (*no-types*, *lifting*) *left5.simps*(*1*) *left6.simps*(*1*) *root-bisim.simps slist.con-rews*(*3*))

lemma *next-grep6-cases*[*consumes 3*, *case-names gl nl*]:
  assumes $vs \neq \bot$
  assumes $xs \neq \bot$
  assumes *P* (*next·xs·*(*grep6·pat·*(*left6·pat·us*)·(*us*, *vs*)))
  obtains (*gl*) *P* (*grep6·pat·*(*left6·pat·us*)·(*us*, *vs*)) | (*nl*) *P* (*next·vs·*(*left6·pat·us*))
using *assms*
apply *atomize-elim*
apply (*subst grep6.unfold*)
apply (*subst* (*asm*) *grep6.unfold*)
apply (*cases xs*; *clarsimp*)
apply (*cases vs*; *clarsimp simp*: *If2-def*[*symmetric*] *split*: *If2-splits*)
done

lemma *root-bisim-op-next56*:
  assumes *root-bisim pat t6 t5*
  assumes $prefix \cdot [:x:] \cdot xs = FF$
  shows $op6 \cdot pat \cdot (next \cdot xs \cdot t6) \cdot x = op6 \cdot pat \cdot t6 \cdot x \wedge op5 \cdot pat \cdot (next \cdot xs \cdot t5) \cdot x = op5 \cdot pat \cdot t5 \cdot x$
using ‹*root-bisim pat t6 t5*›
proof *cases*
  case *Null* **with** *assms*(*2*) **show** *?thesis* **by** (*cases xs*) *simp-all*
next
  case (*gl us vs*) **with** *assms*(*2*) **show** *?thesis*
    apply (*cases* $x = \bot$, *simp*)
    apply (*cases xs*; *clarsimp*)
    apply (*subst* (*1 2*) *grep6.simps*)
    apply (*subst* (*1 2*) *grep5.simps*)
    apply (*cases vs*; *clarsimp simp*: *If2-def*[*symmetric*] *split*: *If2-splits*)
    done
qed *simp*

The main part of establishing that *root-bisim* is a *tree-bisim* is in showing that the left paths constructed by the *grep*s are *root-bisim*-related. We do this by inducting over the length of the pattern so far matched (*us*), as we did when proving that this tree has the 'K' property in §4.2.

lemma
  assumes $pat \neq \bot$
  shows *root-bisim-op*: *root-bisim pat t6 t5* $\Longrightarrow$ *root-bisim pat* (*op6·pat·t6·x*) (*op5·pat·t5·x*) — unused
    and *root-bisim-next-left*: *root-bisim pat* (*next·vs·*(*left6·pat·us*)) (*next·vs·*(*left5·pat·us*)) (**is** *?rbnl us vs*)
proof −
  let $?ogl5 = \lambda us\ vs.\ op5 \cdot pat \cdot (grep5 \cdot pat \cdot (left5 \cdot pat \cdot us) \cdot (us,\ vs)) \cdot x$
  let $?ogl6 = \lambda us\ vs.\ op6 \cdot pat \cdot (grep6 \cdot pat \cdot (left6 \cdot pat \cdot us) \cdot (us,\ vs)) \cdot x$
  let $?for5 = \lambda us.\ sfoldl \cdot (op5 \cdot pat) \cdot (root5 \cdot pat) \cdot us$
  let $?for6 = \lambda us.\ sfoldl \cdot (op6 \cdot pat) \cdot (root6 \cdot pat) \cdot us$
  have $[\![?ogl6\ us\ vs = Node \cdot usvs \cdot l \cdot r;\ cfst \cdot usvs \neq \bot;\ x \neq \bot]\!] \Longrightarrow le \cdot (slength \cdot (cfst \cdot usvs)) \cdot (slength \cdot us + 1) = TT$
   and $[\![op6 \cdot pat \cdot (next \cdot xs \cdot (grep6 \cdot pat \cdot (left6 \cdot pat \cdot us) \cdot (us,\ vs))) \cdot x = Node \cdot usvs \cdot l \cdot r;\ cfst \cdot usvs \neq \bot;\ x \neq \bot;\ us \neq \bot;\ vs$

≠ ⊥⟧ ⟹ le·(slength·(cfst·usvs))·(slength·us + 1) = TT
and ⟦?for6 us = Node·usvs·l·r; cfst·usvs ≠ ⊥⟧ ⟹ lt·(slength·(cfst·usvs))·(slength·us + 1) = TT
and ⟦us ≠ ⊥; vs ≠ ⊥⟧ ⟹ root-bisim pat (?ogl6 us vs) (?ogl5 us vs)
and root-bisim pat (?for6 us) (?for5 us)
and ?rbnl us vs
for usvs l r xs us vs
proof(induct us arbitrary: usvs l r vs x xs rule: op-induct′)
case (step us)
have rbl: root-bisim pat (left6·pat·us) (left5·pat·us)
by (cases us; fastforce intro: step(5) simp: left6.unfold left5.unfold)
{ case (1 usvs l r vs x)
from rbl
have L: le·(slength·(prod.fst usvs′))·(slength·us + 1) = TT
if op6·pat·(next·vs·(left6·pat·us))·x = Node·usvs′·l·r
and cfst·usvs′ ≠ ⊥
and vs ≠ ⊥
for usvs′ l r
proof cases
case bottom with that show ?thesis by simp
next
case Null with that show ?thesis
apply simp
apply (subst (asm) root6.unfold)
apply (subst (asm) grep6.unfold)
apply (fastforce intro: le-plus-1)
done
next
case (gl us″ vs″) show ?thesis
proof(cases us)
case bottom with that show ?thesis by simp
next
case snil with that show ?thesis
apply simp
apply (subst (asm) root6.unfold)
apply (subst (asm) grep6.unfold)
apply clarsimp
done
next
case (scons ush ust)
moreover from that gl scons ‹x ≠ ⊥› have le·(slength·(cfst·usvs′))·(slength·us″ + 1) = TT
apply simp
apply (subst (asm) (2) grep6.unfold)
apply (fastforce dest: step(2, 3)[rotated])
done
moreover from gl scons have lt·(slength·us″)·(slength·(stail·us) + 1) = TT
apply simp
apply (subst (asm) grep6.unfold)
apply (fastforce dest: step(3)[rotated])
done
ultimately show ?thesis
apply clarsimp
apply (metis Integer-le-both-plus-1 Ord-linear-class.le-trans le-iff-lt-or-eq)
done
qed
qed
from 1 show ?case
apply (subst (asm) grep6.unfold)
apply (cases vs; clarsimp simp: If2-def[symmetric] split: If2-splits)

```
      apply (rule L; fastforce)
       apply (metis (no-types, lifting) ab-semigroup-add-class.add-ac(1) fst-conv grep6.simps le-refl-Integer
sappend-snil-id-right slength.simps(2) slength-bottom-iff slength-sappend slist.con-rews(3) tree-injects')
     apply (rule L; fastforce)
     done }
  note slength-ogl = this
  { case (2 usvs l r vs x xs)
    from 2 have xs ≠ ⊥ by clarsimp
    from ‹vs ≠ ⊥› ‹xs ≠ ⊥› 2(1) show ?case
    proof(cases rule: next-grep6-cases)
      case gl with ‹cfst·usvs ≠ ⊥› ‹x ≠ ⊥› show ?thesis using slength-ogl by blast
    next
      case nl
      from rbl show ?thesis
      proof cases
        case bottom with nl ‹cfst·usvs ≠ ⊥› show ?thesis by simp
      next
        case Null with nl ‹us ≠ ⊥› ‹vs ≠ ⊥› show ?thesis
          apply simp
          apply (subst (asm) root6.unfold)
          apply (subst (asm) grep6.unfold)
          apply (clarsimp simp: zero-Integer-def one-Integer-def elim!: slengthE)
          done
      next
        case (gl us'' vs'') show ?thesis
        proof(cases us)
          case bottom with ‹us ≠ ⊥› show ?thesis by simp
        next
          case snil with gl show ?thesis by (subst (asm) grep6.unfold) simp
        next
          case (scons u' us') with 2 nl gl show ?thesis
            apply clarsimp
            apply (subst (asm) (4) grep6.unfold)
            apply clarsimp
            apply (drule step(3)[rotated]; clarsimp)
            apply (drule step(2)[rotated]; clarsimp)
            apply (clarsimp simp: lt-le)
            apply (metis Integer-le-both-plus-1 Ord-linear-class.le-trans)
            done
        qed
      qed
    qed }
  { case (3 usvs l r) show ?case
    proof(cases us rule: srev-cases)
      case snil with 3 show ?thesis
        apply (subst (asm) root6.unfold)
        apply (subst (asm) grep6.unfold)
        apply fastforce
        done
    next
      case (ssnoc u' us')
      then have root-bisim pat (?for6 us') (?for5 us') by (fastforce intro: step(5))
      then show ?thesis
      proof cases
        case bottom with 3 ssnoc show ?thesis by simp
      next
        case Null with 3 ssnoc show ?thesis
          apply simp
```

```
        apply (subst (asm) root6.unfold)
        apply (subst (asm) grep6.unfold)
        apply (clarsimp simp: zero-Integer-def one-Integer-def elim!: slengthE)
        done
    next
      case (gl us'' vs'') with 3 ssnoc show ?thesis
        apply clarsimp
        apply (subst (asm) (2) grep6.unfold)
       apply (fastforce simp: zero-Integer-def one-Integer-def split: if-splits dest!: step(1)[rotated] step(3)[rotated]
elim!: slengthE)
        done
    qed
  qed (use 3 in simp) }
  { case (4 vs x) show ?case
    proof(cases prefix·[:x:]·vs)
      case bottom then show ?thesis
        apply (subst grep6.unfold)
        apply (subst grep5.unfold)
        apply auto
        done
    next
      case TT with ‹pat ≠ ⊥› ‹us ≠ ⊥› show ?thesis
        apply (subst grep6.unfold)
        apply (subst grep5.unfold)
        apply (cases vs; clarsimp simp: op6-left6)
        apply (cases us; clarsimp simp del: left6.simps left5.simps simp add: root5-left5)
     apply (metis (no-types, lifting) op5-sfoldl-left5 root5-left5 root-bisim.simps sappend-bottom-iff slist.con-rews(3)
slist.con-rews(4))
        done
    next
      case FF with ‹pat ≠ ⊥› ‹us ≠ ⊥› show ?thesis
        apply (subst grep6.unfold)
        apply (subst grep5.unfold)
        using rbl
        apply (auto simp: root-bisim-op-next56 elim!: root-bisim.cases intro: root-bisim-root)
        apply (subst (asm) grep6.unfold)
        apply (cases us; fastforce dest: step(3)[rotated] intro: step(4))
        done
    qed }
  { case 5 show ?case
    proof(cases us rule: srev-cases)
      case (ssnoc u' us')
      then have root-bisim pat (?for6 us') (?for5 us') by (fastforce intro: step(5))
      then show ?thesis
      proof cases
        case (gl us'' vs'') with ssnoc show ?thesis
          apply clarsimp
          apply (subst (asm) grep6.unfold)
          apply (fastforce dest: step(3)[rotated] intro: step(4))
          done
      qed (use ‹pat ≠ ⊥› ssnoc root-bisim-root in auto)
    qed (use ‹pat ≠ ⊥› root-bisim-root in auto) }
  { case (6 vs)
    from rbl ‹pat ≠ ⊥› show rbnl: ?rbnl us vs
    proof cases
      case bottom then show ?thesis by fastforce
    next
      case Null then show ?thesis by (cases vs) auto
```

next
case (*gl us'' vs''*) **then show** *?thesis*
apply *clarsimp*
apply (*subst grep5.unfold*)
apply (*subst grep6.unfold*)
apply (*subst* (*asm*) *grep5.unfold*)
apply (*subst* (*asm*) *grep6.unfold*)
apply (*cases us*; *clarsimp*; *cases vs''*; *clarsimp*)
apply (*metis Rep-cfun-strict1 bottom left5.simps*(*2*) *left6.simps*(*2*) *next-snil next-strict*(*1*) *rbl*)
apply (*cases vs*; *clarsimp*)
using *rbl* **apply** (*fastforce dest*: *step*(*3*)[*rotated*] *intro*: *step*(*6*) *simp*: *If2-def*[*symmetric*] *simp del*: *eq-FF split*: *If2-splits*)+
done
qed }
qed
from ‹*pat* ≠ ⊥› *this*(*4*) **show** *root-bisim pat t6 t5* ⟹ *root-bisim pat* (*op6·pat·t6·x*) (*op5·pat·t5·x*)
by (*auto elim*!: *root-bisim.cases intro*: *root-bisim-root*)
show ‹*root-bisim pat* (*next·vs·*(*left6·pat·us*)) (*next·vs·*(*left5·pat·us*))› **by** *fact*
qed

With this result in hand the remainder is technically fiddly but straightforward.

lemmas *tree-bisimI* = *iffD2*[*OF fun-cong*[*OF tree.bisim-def*[*unfolded atomize-eq*]], *rule-format*]

lemma *tree-bisim-root-bisim*:
shows *tree-bisim* (*root-bisim pat*)
proof(*rule tree-bisimI*, *erule root-bisim.cases*, *goal-cases bottom Null Node*)
case (*Node x y us vs*) **then show** *?case*
apply (*subst* (*asm*) *grep5.unfold*)
apply (*subst* (*asm*) *grep6.unfold*)
apply *hypsubst-thin*
apply (*clarsimp simp*: *root-bisim-next-left*)
apply (*cases vs*; *clarsimp*)
apply (*cases us*; *clarsimp simp del*: *left6.simps left5.simps simp add*: *op5-sfoldl-left5 op6-left6*)
apply (*metis* (*no-types*, *lifting*) *root5-left5 root-bisim.simps slist.con-rews*(*3*,*4*))
apply (*metis* (*no-types*, *lifting*) *op5-sfoldl-left5 root-bisim.simps sappend-bottom-iff slist.con-rews*(*3*, *4*))
done
qed *simp-all*

lemma *r6-5*:
shows (*root6·pat*, *op6·pat*) = (*root5·pat*, *op5·pat*)
proof(*cases pat* = ⊥)
case *True*
from *True* **have** *root6·pat* = *root5·pat*
apply (*subst root6.unfold*)
apply (*subst grep6.unfold*)
apply (*subst root5.unfold*)
apply (*subst grep5.unfold*)
apply *simp*
done
moreover
from *True* ‹*root6·pat* = *root5·pat*› **have** *op6·pat·t·x* = *op5·pat·t·x* **for** *t x*
by (*induct t*) *simp-all*
ultimately show *?thesis* **by** (*simp add*: *cfun-eq-iff*)
next
case *False*
then have *root*: *root6·pat* = *root5·pat*
by (*rule tree.coinduct*[*OF tree-bisim-root-bisim root-bisim-root*])
moreover

from *root* **have** *op6·pat·t·x* = *op5·pat·t·x* **for** *t x* **by** (*induct t*) *simp-all*
ultimately show *?thesis* **by** (*simp add*: *cfun-eq-iff*)
qed

We conclude this section by observing that accumulator-introduction is a well known technique (see, for instance, Hutton (2016, §13.6)), but the examples in the literature assume that the type involved is defined inductively. Bird adopts this strategy without considering what the mixed inductive/coinductive rule is that justifies the preservation of total correctness.

The difficulty of this step is why we wired in the 'K' opt earlier: it allows us to preserve the shape of the tree all the way from the data refinement to the final version.

4.7 Step 7: Simplify, unfold prefix

The next step (Bird, bottom of p132) is to move the case split in *grep6* on *vs* above the *Node* constructor, which makes *grep7* strict in that parameter and therefore not extensionally equal to *grep6*. We establish a weaker correspondence using fixed-point induction.

We also unfold *prefix* in *op6*.

fixrec
root7 :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) *tree*
and *op7* :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
and *grep7* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*
where
[*simp del*]:
root7·pat = *grep7·pat·Null·*([::], *pat*)
| *op7·pat·Null·x* = *root7·pat*
| *op7·pat·*(*Node·*(*us*, [::])·*l·r*)·*x* = *op7·pat·l·x* — Unfold *prefix*
| ⟦*v* ≠ ⊥; *vs* ≠ ⊥⟧ ⟹
op7·pat·(*Node·*(*us*, *v* :# *vs*)·*l·r*)·*x* = *If eq·x·v then r else op7·pat·l·x*
| [*simp del*]:
grep7·pat·l·(*us*, [::]) = *Node·*(*us*, [::])·*l·Null* — Case split on *vs* hoisted above *Node*.
| ⟦*v* ≠ ⊥; *vs* ≠ ⊥⟧ ⟹
grep7·pat·l·(*us*, *v* :# *vs*) = *Node·*(*us*, *v* :# *vs*)·(*next·*(*v* :# *vs*)·*l*)·(*grep7·pat·*(*op7·pat·l·v*)·(*us* :@ [:*v*:], *vs*))

schematic-goal *root7-op7-grep7-def*:
(*root7* :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) *tree*
, *op7* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
, *grep7* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*)
= *fix·?F*
unfolding *root7-def op7-def grep7-def* **by** *simp*

lemma *r7-6-aux*:
assumes *pat* ≠ ⊥
shows
(Λ (*root*, *op*, *grep*). (*root·pat*, *seq·x·*(*op·pat·t·x*), *grep·pat·l·*(*us*, *vs*)))·
(*root7* :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) *tree*
, *op7* :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
, *grep7* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*)
= (Λ (*root*, *op*, *grep*). (*root·pat*, *seq·x·*(*op·pat·t·x*), *seq·vs·*(*grep·pat·l·*(*us*, *vs*))))·
(*root6* :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) *tree*
, *op6* :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
, *grep6* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*)
unfolding *root6-op6-grep6-def root7-op7-grep7-def*
proof(*induct arbitrary*: *t x l us vs rule*: *parallel-fix-ind*[*case-names adm bottom step*])
case (*step X Y t x l us vs*) **then show** *?case*
apply −
apply (*cases X*, *cases Y*)
apply (*rename-tac r10 o10 g10 r9 o9 g9*)

apply (*clarsimp simp*: *cfun-eq-iff assms match-Node-mplus-match-Node match-Null-match-Node-tree-case tree-case-distr match-snil-match-scons-slist-case slist-case-distr If-distr*)
apply (*intro allI conjI*)
apply (*case-tac t*; *clarsimp*)
apply (*rename-tac us vs l r*)
apply (*case-tac* $x = \bot$; *clarsimp*)
apply (*case-tac vs*; *clarsimp*; *fail*)
apply (*case-tac vs*; *clarsimp*)
apply (*metis ID1 seq-simps*(*3*))
done
qed *simp-all*

lemma *r7-6*:
assumes $pat \neq \bot$
shows $root7{\cdot}pat = root6{\cdot}pat$
and $x \neq \bot \Longrightarrow op7{\cdot}pat{\cdot}t{\cdot}x = op6{\cdot}pat{\cdot}t{\cdot}x$
using *r7-6-aux*[*OF assms*] **by** (*force simp*: *cfun-eq-iff dest*: *spec*[**where** $x{=}x$])+

4.8 Step 8: Discard us

We now discard *us* from the tree as it is no longer used. This requires a new definition of *next*. This is essentially another data refinement.

fixrec $next'$ :: $'a{::}Eq\text{-}def \to [:'a:]\ tree \to [:'a:]\ tree$ **where**
$next'{\cdot}x{\cdot}Null = Null$
$|\ next'{\cdot}x{\cdot}(Node{\cdot}[::]{\cdot}l{\cdot}r) = Node{\cdot}[::]{\cdot}l{\cdot}r$
$|\ [\![v \neq \bot;\ vs \neq \bot;\ x \neq \bot]\!] \Longrightarrow$
$next'{\cdot}x{\cdot}(Node{\cdot}(v :\#\ vs){\cdot}l{\cdot}r) = If\ eq{\cdot}x{\cdot}v\ then\ l\ else\ Node{\cdot}(v :\#\ vs){\cdot}l{\cdot}r$

fixrec — Bird p133
$root8$ :: $[:'a{::}Eq\text{-}def:] \to [:'a:]\ tree$
and $op8$ :: $[:'a:] \to [:'a:]\ tree \to\ 'a \to [:'a:]\ tree$
and $grep8$ :: $[:'a:] \to [:'a:]\ tree \to [:'a:] \to [:'a:]\ tree$
where
[*simp del*]:
$root8{\cdot}pat = grep8{\cdot}pat{\cdot}Null{\cdot}pat$
$|\ op8{\cdot}pat{\cdot}Null{\cdot}x = root8{\cdot}pat$
$|\ op8{\cdot}pat{\cdot}(Node{\cdot}[::]{\cdot}l{\cdot}r){\cdot}x = op8{\cdot}pat{\cdot}l{\cdot}x$
$|\ [\![v \neq \bot;\ vs \neq \bot]\!] \Longrightarrow$
$op8{\cdot}pat{\cdot}(Node{\cdot}(v :\#\ vs){\cdot}l{\cdot}r){\cdot}x = If\ eq{\cdot}x{\cdot}v\ then\ r\ else\ op8{\cdot}pat{\cdot}l{\cdot}x$
$|\ grep8{\cdot}pat{\cdot}l{\cdot}[::] = Node{\cdot}[::]{\cdot}l{\cdot}Null$
$|\ [\![v \neq \bot;\ vs \neq \bot]\!] \Longrightarrow$
$grep8{\cdot}pat{\cdot}l{\cdot}(v :\#\ vs) = Node{\cdot}(v :\#\ vs){\cdot}(next'{\cdot}v{\cdot}l){\cdot}(grep8{\cdot}pat{\cdot}(op8{\cdot}pat{\cdot}l{\cdot}v){\cdot}vs)$

fixrec $ok8$ :: $[:'a:]\ tree \to tr$ **where**
$vs \neq \bot \Longrightarrow ok8{\cdot}(Node{\cdot}vs{\cdot}l{\cdot}r) = snull{\cdot}vs$

schematic-goal *root8-op8-grep8-def*:
($root8$ :: $[:'a{::}Eq\text{-}def:] \to [:'a:]\ tree$
, $op8$ :: $[:'a:] \to [:'a:]\ tree \to\ 'a \to [:'a:]\ tree$
, $grep8$ :: $[:'a:] \to [:'a:]\ tree \to [:'a:] \to [:'a:]\ tree$)
$= fix{\cdot}?F$
unfolding *op8-def root8-def grep8-def* **by** *simp*

lemma *next′-strict*[*simp*]:
$next'{\cdot}x{\cdot}\bot = \bot$
$next'{\cdot}\bot{\cdot}(Node{\cdot}(v :\#\ vs){\cdot}l{\cdot}r) = \bot$
by (*cases* $v :\#\ vs = \bot$; *fixrec-simp*; *clarsimp*)+

lemma *root8-op8-grep8-strict*[*simp*]:
  grep8·pat·l·⊥ = ⊥
  op8·pat·⊥ = ⊥
  root8·⊥ = ⊥
by *fixrec-simp*+

lemma *ok8-strict*[*simp*]:
  ok8·⊥ = ⊥
  ok8·Null = ⊥
by *fixrec-simp*+

We cannot readily relate *next* and *next′* using worker/wrapper as the obvious abstraction is not invertible. Conversely the desired result is easily shown by fixed-point induction.

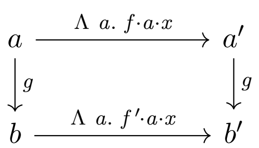
lemma *next′-next*:
  assumes *v* ≠ ⊥
  assumes *vs* ≠ ⊥
  shows *next′·v*·(*tree-map′·csnd·t*) = *tree-map′·csnd*·(*next*·(*v* :# *vs*)·*t*)
proof(*induct t*)
  case (*Node usvs′ l r*) **with** *assms* **show** *?case*
    apply (*cases usvs′*; *clarsimp*)
    apply (*rename-tac us″ vs″*)
    apply (*case-tac vs″*; *clarsimp simp*: *If-distr*)
    done
qed (*use assms* **in** *simp-all*)

lemma *r8-7*[*simplified*]:
  shows (Λ (*root*, *op*, *grep*). (*root·pat*
      , *op·pat*·(*tree-map′·csnd·t*)·*x*
      , *grep·pat*·(*tree-map′·csnd·l*)·(*csnd·usvs*)))·(*root8*, *op8*, *grep8*)
    = (Λ (*root*, *op*, *grep*). (*tree-map′·csnd*·(*root·pat*)
      , *tree-map′·csnd*·(*op·pat·t·x*)
      , *tree-map′·csnd*·(*grep·pat·l·usvs*)))·(*root7*, *op7*, *grep7*)
unfolding *root7-op7-grep7-def root8-op8-grep8-def*
proof(*induct arbitrary*: *l t usvs x rule*: *parallel-fix-ind*[*case-names adm bottom step*])
  case (*step X Y l t usvs x*) **then show** *?case*
    apply −
    apply (*cases X*; *cases Y*)
    apply (*clarsimp simp*: *cfun-eq-iff next′-next*
        match-snil-match-scons-slist-case slist-case-distr
        match-Node-mplus-match-Node match-Null-match-Node-tree-case tree-case-distr
      cong: *slist-case-cong*)
    apply (*cases t*; *clarsimp simp*: *If-distr*)
    apply (*rename-tac us vs l r*)
    apply (*case-tac vs*; *fastforce*)
    done
qed *simp-all*

Top-level equivalence follows from *lfp-fusion* specialized to *sscanl* (*sscanl-lfp-fusion*), which states that

$$smap \cdot g \ oo \ sscanl \cdot f \cdot z = sscanl \cdot f' \cdot (g \cdot z)$$

provided that *g* is strict and the following diagram commutes for $x \neq \bot$:



lemma *ok8-ok8*: *ok8 oo tree-map′·csnd* = *snull oo csnd oo abs2* (**is** *?lhs* = *?rhs*)
proof(*rule cfun-eqI*)
  fix *t* **show** *?lhs·t* = *?rhs·t*
    by (*cases t*; *clarsimp*) (*metis ok8.simps ok8-strict*(*1*) *snull-strict tree.con-rews*(*3*))
qed

lemma *matches8*: — Bird p133
  shows *matches·pat* = *smap·cfst oo sfilter·*(*ok8 oo csnd*) *oo sscanl·*(Λ (*n*, *x*) *y*. (*n* + *1*, *op8·pat·x·y*))·(*0*, *root8·pat*)
(**is** *?lhs* = *?rhs*)
proof(*cases pat* = ⊥)
  case *True*
  then have *?lhs·xs* = *?rhs·xs* **for** *xs* **by** (*cases xs*; *clarsimp*)
  then show *?thesis* **by** (*simp add*: *cfun-eq-iff*)
next
  case *False*
  then have ∗: *matches·pat* = *smap·cfst oo sfilter·*(*snull oo csnd oo abs2 oo csnd*) *oo sscanl·*(Λ (*n*, *x*) *y*. (*n* + *1*, *op7·pat·x·y*))·(*0*, *root7·pat*)
    using *data-refinement*[**where** *′a*=*′a*] *r3-2*[**where** *′a*=*′a*] *r4-3*[**where** *′a*=*′a*] *r5-4*[**where** *′a*=*′a*] *r6-5*(*1*)[**where** *pat*=*pat*] *r7-6*[**where** *pat*=*pat*]
    unfolding *matches2.unfold* **by** (*fastforce simp*: *oo-assoc cfun-eq-iff csplit-def intro*!: *cfun-arg-cong sscanl-cong*)
  from ‹*pat* ≠ ⊥› **show** *?thesis*
    apply −
    apply (*subst conjunct1*[*OF r8-7*])
    apply (*subst sscanl-lfp-fusion*[**where** *g*=*ID* ∗∗ *tree-map′·csnd* **and** *z*=(*0*, *root7·pat*), *simplified*, *symmetric*])
     prefer *2*
     apply (*subst oo-assoc*, *subst sfilter-smap*, *simp*)
     apply (*simp add*: *oo-assoc*)
     apply (*simp add*: *oo-assoc*[*symmetric*])
     apply (*subst oo-assoc*, *subst ok8-ok8*)
     apply (*clarsimp simp*: *oo-assoc* ∗)
     apply (*rule refl*)
    apply (*clarsimp simp*: *r8-7*)
    done
qed

4.9 Step 9: Factor out pat (final version)

Finally we factor *pat* from these definitions and arrive at Bird's cyclic data structure, which when executed using lazy evaluation actually memoises the computation of *grep8*.

The *Letrec* syntax groups recursive bindings with , and separates these with ;. Its lack of support for clausal definitions, and that HOLCF *case* does not support nested patterns, leads to some awkwardness.

fixrec *matchesf* :: [:*′a*::*Eq-def*:] → [:*′a*:] → [:*Integer*:] **where**
[*simp del*]: *matchesf·pat* =
    (*Letrec okf* = (Λ (*Node·vs·l·r*). *snull·vs*);
            nextf = (Λ *x t*. *case t of*
                    Null ⇒ *Null*
                  | *Node·vs·l·r* ⇒ (*case vs of*
                      [::] ⇒ *t*
                    | *v* :# *vs* ⇒ *If eq·x·v then l else t*));
            rootf = *grepf·Null·pat*,
            opf = (Λ *t x*. *case t of*
                    Null ⇒ *rootf*
                  | *Node·vs·l·r* ⇒ (*case vs of*
                      [::] ⇒ *opf·l·x*
                    | *v* :# *vs* ⇒ *If eq·x·v then r else opf·l·x*)),
            grepf = (Λ *l vs*. *case vs of*
                    [::] ⇒ *Node*·[::]·*l*·*Null*

```
            | v :# vs ⇒ Node·(v :# vs)·(nextf·v·l)·(grepf·(opf·l·v)·vs));
         stepf = (Λ (n, t) x. (n + 1, opf·t·x))
       in smap·cfst oo sfilter·(okf oo csnd) oo sscanl·stepf·(0, rootf))
```

```
lemma matchesf-ok8: (Λ (Node·vs·l·r). snull·vs) = ok8
by (clarsimp simp: cfun-eq-iff; rename-tac x; case-tac x; clarsimp)
```

```
lemma matchesf-next':
  (Λ x t. case t of Null ⇒ Null | Node·vs·l·r ⇒ (case vs of [::] ⇒ t | v :# vs ⇒ If eq·x·v then l else t)) = next'
apply (clarsimp simp: cfun-eq-iff next'.unfold
                      match-snil-match-scons-slist-case slist-case-distr
                      match-Node-mplus-match-Node match-Null-match-Node-tree-case tree-case-distr)
apply (simp cong: tree-case-cong)
apply (simp cong: slist-case-cong)
done
```

```
lemma matchesf-8:
  fix·(Λ (Rootf, Opf, Grepf).
        ( Grepf·Null·pat
        , Λ t x. case t of Null ⇒ Rootf | Node·vs·l·r ⇒
              (case vs of [::] ⇒ Opf·l·x | v :# vs ⇒ If eq·x·v then r else Opf·l·x)
        , Λ l vs. case vs of [::] ⇒ Node·[::]·l·Null | v :# vs ⇒ Node·(v :# vs)·(next'·v·l)·(Grepf·(Opf·l·v)·vs)) )
= (Λ (root, op, grep). (root·pat, op·pat, grep·pat))·(root8, op8, grep8)
unfolding root8-op8-grep8-def
by (rule lfp-fusion[symmetric])
   (fastforce simp: cfun-eq-iff
                    match-snil-match-scons-slist-case slist-case-distr
                    match-Node-mplus-match-Node match-Null-match-Node-tree-case tree-case-distr)+
```

```
theorem matches-final:
  shows matches = matchesf
by (clarsimp simp: cfun-eq-iff fix-const eta-cfun csplit-cfun3 CLetrec-def
                   matches8 matchesf.unfold matchesf-next' matchesf-ok8 matchesf-8[simplified eta-cfun])
```

The final program above is easily syntactically translated into the Haskell shown in Figure 1, and one can expect GHC's list fusion machinery to compile the top-level driver into an efficient loop. Lochbihler and Maximova (2015) have mechanised this optimisation for Isabelle/HOL's code generator (and see also Huffman (2009)).

As we lack both pieces of infrastructure we show such a fusion is sound by hand.

```
lemma fused-driver':
  assumes g·⊥ = ⊥
  assumes p·⊥ = ⊥
  shows smap·g oo sfilter·p oo sscanl·f·z
     = (μ R. Λ z xxs. case xxs of
                 [::] ⇒ If p·z then [:g·z:] else [::]
               | x :# xs ⇒ let z' = f·z·x in If p·z then g·z :# R·z'·xs else R·z'·xs)·z
(is ?lhs = ?rhs)
proof(rule cfun-eqI)
  fix xs from assms show ?lhs·xs = ?rhs·xs
    by (induct xs arbitrary: z) (subst fix-eq; fastforce simp: If-distr Let-def)+
qed
```

5 Related work

Derivations of KMP matching are legion and we do not attempt to catalogue them here.

Bird and colleagues have presented versions of this story at least four times. All treat MP, not KMP (see §4.2), and use a style of equational reasoning with fold/unfold transformations (Burstall and Darlington 1977) that only establishes partial correctness (see §1.1). Briefly:

- The second example of Bird (1977) is an imperative program that is similar to MP.
- Bird et al. (1989) devised the core of the derivation mechanized here, notably omitting a formal justification for the final data refinement step that introduces the circular data structure.
- Bird (2005) refines Bird et al. (1989) and derives Boyer-Moore matching (Gusfield 1997, §2.2) in a similar style.
- Bird (2010, Chapter 17) further refines Bird (2005) and is the basis of the work discussed here. Bird (2012, §3.1) contains some further relevant remarks.

Ager et al. (2006) show how KMP matchers (specialised to a given pattern) can be derived by the partial evaluation of an initial program in linear time. We observe that neither their approach, of incorporating the essence of KMP in their starting point, nor Bird's of introducing it by data refinement (§4.2), provides a satisfying explanation of how KMP could be discovered; Pottier (2012) attempts to do this. In contrast to Bird, these and most other presentations make heavy use of arrays and array indexing which occludes the central insights.

6 Implementations

With varying amounts of effort we can translate our final program of §4.9 into a variety of languages. The most direct version, in Haskell, was shown in Figure 1. An ocaml version is similar due to that language's support for laziness. In contrast Standard ML requires an encoding; we use backpatching as shown in Figure 4. In both cases the tree datatype can be made strict in the right branch as it is defined by primitive recursion on the pattern.

More interestingly, our derivation suggests that Bird's KMP program can be computed using *rational* trees (also known as *regular* trees (Courcelle 1983)), which are traditionally supported by Prolog implementations. Our version is shown in Figure 3. This demonstrates that the program could instead be thought of as a computation over difference structures. Colmerauer (1982); Giannesini and Cohen (1984) provide more examples of this style of programming. We leave a proof of correctness to future work.

7 Concluding remarks

Our derivation leans heavily on domain theory's ability to reason about partially-defined objects that are challenging to handle at present in a language of total functions. Conversely it is too abstract to capture the operational behaviour of the program as it does not model laziness. It would also be interesting to put the data refinement of §4.2 on a firmer foundation by deriving the memoizing datatype from the direct program of §4.1. Haskell fans may care to address the semantic discrepancies mentioned in §1.1.

References

M. S. Ager, O. Danvy, and H. K. Rohde. Fast partial evaluation of pattern matching in strings. *ACM Transactions on Programming Languages and Systems*, 28(4):696–714, 2006. doi: 10.1145/1146812.

R. S. Bird. Improving programs by the introduction of recursion. *Communications of the ACM*, 20(11):856–863, 1977. doi: 10.1145/359863.359889.

R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987. NATO ASI Series F Volume 36. Also available as Technical Monograph PRG-56, from the Programming Research Group, Oxford University.

R. S. Bird. Polymorphic string matching. In *Haskell'2005*, pages 110–115. ACM, 2005. doi: 10.1145/1088348.1088359.

R. S. Bird. *Pearls of Functional Algorithm Design*. CUP, 2010. ISBN 9780521513388.

R. S. Bird. On building cyclic and shared structures in Haskell. *Formal Aspects of Computing*, 24(4-6):609–621, 2012. doi: 10.1007/s00165-012-0243-6.

```
% -*- mode: prolog -*-
% Bird's Morris-Pratt string matcher
% Chapter 17, "Pearls of Functional Algorithm Design", 2010.
%   - adapted to use rational trees.
%   - with the ‘K’ (next) optimisation
% Tested with SWI Prolog, which has good support for rational trees.

% root/2 (+, -) det
root(Ws, T) :- grep(T, null, Ws, T).

% op/4 (?, +, +, -) det <-- Root may or may not be fully ground
op(Root, null,                _X, Root).
op(Root, node([],      L, _R), X, T) :- op(Root, L, X, T).
op(Root, node([V|_Vs], L,  R), X, T) :-
    (X = V -> T = R ; op(Root, L, X, T)).

% next/3 (+, +, -) det
next(_X, null,               null).
next(_X, node([], L, R),     node([], L, R)).
next( X, node([V|Vs], L, R), T) :- ( X = V -> T = L ; T = node([V|Vs], L, R) ).

% grep/4 (+, +, +, -) det
grep(_Root, L, [],     node([], L, null)).
grep( Root, L, [V|Vs], node([V|Vs], L1, R)) :-
    next(V, L, L1), op(Root, L, V, T), grep(Root, T, Vs, R).

% ok/1 (+) det
ok(node([],  _L, _R)).

%% Driver

% matches_aux/5 (+, +, +, +, -) det
matches_aux(_Root, N, T, [],     Ns) :- ( ok(T) -> Ns = [N] ; Ns = [] ).
matches_aux( Root, N, T, [X|Xs], Ns) :-
    N1 is N + 1, op(Root, T, X, T1),
    ( ok(T) -> ( Ns = [N|Ns1], matches_aux(Root, N1, T1, Xs, Ns1) )
             ; matches_aux(Root, N1, T1, Xs, Ns) ).

% matches/3 (+, +, -) det
matches(Ws, Txt, Ns) :- root(Ws, Root), matches_aux(Root, 0, Root, Txt, Ns).

% :- root([1,2,1], Root).
% :- root([1,2,1,1,2], Root).
% :- matches([1,2,3,1,2], [1,2,1,2,3,1,2,3,1,2], Ns).
```

Figure 3: The final KMP program transliterated into Prolog.

```
(* Bird's Morris-Pratt string matcher
   Chapter 17, "Pearls of Functional Algorithm Design", 2010.
    - with the 'K' (next) optimisation
    - using backpatching
 *)

structure KMP :> sig val kmatches : ('a * 'a -> bool) -> 'a list -> 'a list -> int list end =
struct

datatype 'a thunk = Val of 'a | Thunk of unit -> 'a

type 'a lazy = 'a thunk ref

fun lazy (f: unit -> 'a) : 'a lazy =
  ref (Thunk f)

fun force (su : 'a lazy) : 'a =
  case !su of
      Val v => v
    | Thunk f => let val v = f () in su := Val v; v end

datatype 'a tree
  = Null
  | Node of 'a list * 'a tree lazy * 'a tree

type 'a ltree = 'a tree lazy

fun kmatches (eq: 'a * 'a -> bool) (ws: 'a list) : 'a list -> int list =
  let
    fun ok (t: 'a ltree) : bool = case force t of Node ([], l, r) => true | _ => false
    fun next (x: 'a) (t: 'a ltree) : 'a ltree =
           lazy (fn () => let val t = force t in case t of
             Null => Null
           | Node ([], _, _) => t
           | Node (v :: vs, l, _) => if eq (x, v) then force l else t end)
     (* Backpatching! *)
     val root : 'a ltree = lazy (fn () => raise Fail "blackhole")
     fun op' (t: 'a ltree) (x: 'a) : 'a ltree =
           lazy (fn () => case force t of
             Null => force root
           | Node (vvs, l, r) =>
             (case vvs of
                [] => force (op' l x)
              | v :: vs => if eq (x, v) then r else force (op' l x)))
     and grep (l: 'a ltree) (vvs: 'a list): 'a tree =
           ( (* print "grep: produce node\n"; *) case vvs of
             [] => Node ([], l, Null)
           | v :: vs => Node (vvs, next v l, grep (op' l v) vs) )
     val () = root := Thunk (fn () => grep (lazy (fn () => Null)) ws)
     fun step ((n, t): int * 'a ltree) (x: 'a) : int * 'a ltree = (n + 1, op' t x)
     fun rheight (t: 'a tree) =
           case t of Null => 0 | Node (_, _, r) => 1 + rheight r
     fun driver ((n, t): int * 'a ltree) (xxs: 'a list) : int list =
           case xxs of
             [] => if ok t then [n] else []
           | x :: xs => let val nt' = step (n, t) x
                        in if ok t then n :: driver nt' xs else driver nt' xs end
  in
    driver (0, root)
  end

end;
```

Figure 4: The final KMP program transliterated into Standard ML.

R. S. Bird, J. Gibbons, and G. Jones. Formal derivation of a pattern matching algorithm. *Science of Computer Programming*, 12(2):93–104, 1989. doi: 10.1016/0167-6423(89)90036-1.

J. C. Blanchette, A. Bouzy, A. Lochbihler, A. Popescu, and D. Traytel. Friends with benefits - implementing corecursion in foundational proof assistants. In *ESOP'2017*, volume 10201 of *LNCS*, pages 111–140. Springer, 2017. doi: 10.1007/978-3-662-54434-1_5.

R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977. doi: 10.1145/321992.321996.

A. Colmerauer. Prolog and infinite trees. In K. L. Clark and S. Å. Tarnlund, editors, *Logic Programming*, pages 107–114. Academic Press, 1982.

B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25:95–169, 1983. doi: 10.1016/0304-3975(83)90059-2.

M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific, 2002. ISBN 981-02-4782-6.

P. Gammie. Short note: Strict unwraps make worker/wrapper fusion totally correct. *Journal of Functional Programming*, 21:209–213, 2011.

F. Giannesini and J. Cohen. Parser generation and grammar manipulation using Prolog's infinite trees. *Journal of Logic Programming*, 1(3):253–265, 1984. doi: 10.1016/0743-1066(84)90013-X.

A. Gill and G. Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009.

D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* CUP, 1997. ISBN 0-521-58519-8.

R. Hinze and J. Jeuring. Weaving a web. *Journal of Functional Programming*, 11(6):681–689, 2001. doi: 10.1017/S0956796801004129.

B. Huffman. Stream fusion. *Archive of Formal Proofs*, April 2009. URL http://isa-afp.org/entries/Stream-Fusion.html.

G. Hutton. *Programming in Haskell.* CUP, second edition, September 2016.

J.-B. Jeannin, D. Kozen, and A. Silva. Cocaml: Functional programming with regular coinductive types. *Fundamenta Informaticae*, 150(3-4):347–377, 2017. doi: 10.3233/FI-2017-1473.

D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6 (2):323–350, 1977. doi: 10.1137/0206024.

A. Lochbihler and A. Maximova. Stream fusion for Isabelle's code generator - rough diamond. In *ITP'2015*, volume 9236 of *LNCS*, pages 270–277. Springer, 2015. doi: 10.1007/978-3-319-22102-1_18.

O. Müller, T. Nipkow, D. von Oheimb, and O. Slotosch. HOLCF = HOL + LCF. *Journal of Functional Programming*, 9:191–223, 1999.

L. C. Paulson. *Logic and computation - interactive proof with Cambridge LCF*, volume 2 of *Cambridge Tracts in Theoretical Computer Science.* CUP, 1987. ISBN 978-0-521-34632-0.

F. Pottier. Reconstructing the Knuth-Morris-Pratt algorithm, 2012. URL http://gallium.inria.fr/blog/kmp/.

M. Takeichi and Y. Akama. Deriving a functional Knuth-Morris-Pratt algorithm by transformation. *Journal of Information Processing*, 13(4):522–528, April 1991.

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.

J. van der Woude. Playing with patterns, searching for strings. *Science of Computer Programming*, 12(3):177–190, 1989. doi: 10.1016/0167-6423(89)90001-4.