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

Peter Gammie

February 23, 2021

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
  1.1 Formal setting .................................................. 3

2 Extra HOLCF .......................................................... 3
  2.1 Extra HOLCF Prelude. ........................................... 6
  2.2 Element equality .................................................. 6
  2.3 Recursive let bindings .......................................... 7

3 Strict lists ............................................................ 7
  3.1 Some of the usual reasoning infrastructure .................. 10
  3.2 Some of the usual operations .................................. 11

4 Knuth-Morris-Pratt matching according to Bird ................. 28
  4.1 Step 1: Specification ............................................ 28
  4.2 Step 2: Data refinement and the ‘K’ optimisation ........... 35
  4.3 Step 3: Introduce an accumulating parameter (grep) ........ 41
  4.4 Step 4: Inline rep ............................................... 42
  4.5 Step 5: Simplify to Bird’s “simpler forms” .................. 43
  4.6 Step 6: Memoize left ............................................. 44
  4.7 Step 7: Simplify, unfold prefix ................................ 51
  4.8 Step 8: Discard us ............................................. 52
  4.9 Step 9: Factor out pat (final version) ....................... 54

5 Related work ................................................................ 55

6 Implementations ......................................................... 56

7 Concluding remarks .................................................... 56

References .................................................................. 59

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

matches :: Eq a => [a] -> [a] -> [Integer]
movies 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

        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 ⊑ 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 = 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·⊥ = ⊥
assumes g oo f = h oo g
shows g·(fix·f) = fix·h
proof (induct rule: parallel-fix-ind)
case 2 show g·⊥ = ⊥ by fact
case (3 x y)
from ⟨g·x = y⟩ ⟨g oo f = h oo g⟩ show g·(f·x) = h·y
  by (simp add: cfun-eq-iff)
qed simp

lemma predE:
  obtains (strict) p·⊥ = ⊥ | (FF) p = (Λ x. FF) | (TT) p = (Λ x. TT)
using flat-codom[where f=p and x=⊥] by (cases p·⊥; force simp: cfun-eq-iff)

lemma retraction-cfcomp-strict:
\textbf{assumes} \( f \circ \circ g = \text{ID} \)
\textbf{shows} \( f \cdot \bot = \bot \)
\textbf{using} \texttt{assms retraction-strict by (clarsimp simp: cfun-eq-iff)}

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

\textbf{lemmas} \texttt{oo-assoc = assoc-oo} — Normalize name

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

\textbf{lemma} \texttt{seq-below[iff]}: \texttt{seq} \cdot x \subseteq y
\textbf{by} \texttt{(simp add: seq-conv-if)}

\textbf{lemma} \texttt{seq-strict-distr}: \texttt{f} \cdot \bot = \bot = \Rightarrow \texttt{seq} \cdot x \cdot (\texttt{f} \cdot y) = \texttt{f} \cdot (\texttt{seq} \cdot x \cdot y)
\textbf{by} \texttt{(cases \( x = \bot \); clarsimp)}

\textbf{lemma} \texttt{strictify-below[iff]}: \texttt{strictify} \cdot f \subseteq f
\textbf{unfolding} \texttt{strictify-def} \textbf{by} \texttt{(clarsimp simp: cfun-below-iff)}

\textbf{lemma} \texttt{If-distr}: \\
\( f \cdot \bot = \bot = \Rightarrow f \cdot (\texttt{If \( b \) then \( t \) else \( e \)}) = (\texttt{If \( b \) then \( f \cdot t \) else \( f \cdot e \)}) \)
\textbf{by} \texttt{(case-tac \(!\) \( b \); simp-all)}

\textbf{lemma} \texttt{If-tr}: \( (b \cdot \bot = \bot = \Rightarrow (b \cdot \text{andalso} \ e) = \texttt{If \( b \) then \( t \) else \( e \cdot \bot \))} \)
\textbf{by} \texttt{(cases \( b \); simp-all)}

\textbf{lemma} \texttt{If-andalso}: \( \texttt{If \( p \) andalso \( q \) then \( t \) else \( e \) = \texttt{If \( p \) then \( f \) then \( t \) else \( e \) else \( e \)} } \)
by (cases p) simp-all

lemma If-else-absorb:
  assumes c = ⊥ → e = ⊥
  assumes c = TT → e = t
  shows If c then t else e = e
using assms by (cases c; clarsimp)

lemma andalso-cong: [P = P'; P' = TT → Q = Q'] → (P andalso Q) = (P' andalso Q')
by (cases P) simp-all

lemma andalso-weaken-left:
  assumes P = TT → Q = TT
  assumes P = FF → Q ≠ ⊥
  assumes P = ⊥ → Q ≠ FF
  shows P = (Q andalso P)
using assms by (cases P; cases Q; simp)

lemma orelse-cong: [P = P'; P' = FF → Q = Q'] → (P orelse Q) = (P' orelse Q')
by (cases P) simp-all

lemma orelse-conv[simp]:
  ((x orelse y) = TT) ←→ (x = TT ∨ (x = FF ∧ y = TT))
  ((x orelse y) = ⊥) ←→ (x = ⊥ ∨ (x = FF ∧ y = ⊥))
by (cases x; cases y; simp)+

lemma csplit-cfun2: cont F ⇒ (Λ x. F x) = (Λ (x, y). F (x, y))
by (clarsimp simp: cfun-eq-iff prod-cont-iff)

lemma csplit-cfun3: cont F ⇒ (Λ x. F x) = (Λ (x, y, z). F (x, y, z))
by (clarsimp simp: cfun-eq-iff prod-cont-iff)

definition convol :: ('a::cpo → 'b::cpo) → ('a → 'c::cpo) → 'a → 'b × 'c where
  convol = (Λ f g x. (f⋅x, g⋅x))
abbreviation convol-syn :: ('a::cpo → 'b::cpo) ⇒ ('a → 'c::cpo) ⇒ 'a → 'b × 'c (infix && 65) where
  f && g ≡ convol⋅f⋅g

lemma convol-strict[simp]:
  convol⋅⊥⋅⊥ = ⊥
unfolding convol-def by simp

lemma convol-simp[simp]: (f && g)⋅x = (f⋅x, g⋅x)
unfolding convol-def by simp

definition map-prod :: ('a::cpo → 'c::cpo) → ('b::cpo → 'd) → 'a × 'b → 'c × 'd where
  map-prod = (Λ f g (x, y). (f⋅x, g⋅y))
abbreviation map-prod-syn :: ('a → 'c) ⇒ ('b → 'd) ⇒ 'a × 'b → 'c × 'd (infix ** 65) where
  f ** g ≡ map-prod⋅f⋅g

lemma map-prod-cfcomp[simp]: (f ** m) oo (g ** n) = (f oo g) ** (m 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)⋅x = (f⋅(cfst⋅x), g⋅(csnd⋅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 · (⊥ :: 'a::Eq-strict) = ⊥
by (simp add: cfun-eq-iff)

lemma Integer-le-both-plus-1[simp]:
fixes m :: Integer
shows le · (m + 1) · (n + 1) = le · m · n
by (cases m; cases n; simp add: one-Integer-def)

lemma plus-eq-MkI-conv:
l + n = MkI · m ←→ (∃ l′ n′. l = MkI · l′ ∧ n = MkI · n′ ∧ m = l′ + n′)
by (cases l; simp) (cases n; auto)

lemma lt-defined:
fixes x :: Integer
shows lt · x · y = TT =⇒ (∃ x ≠ ⊥ ∧ y ≠ ⊥)
lt · x · y = FF =⇒ (∃ x ≠ ⊥ ∧ y ≠ ⊥)
by (cases x; cases y; clarsimp) +

lemma le-defined:
fixes x :: Integer
shows le · x · y = TT =⇒ (∃ x ≠ ⊥ ∧ y ≠ ⊥)
le · x · y = FF =⇒ (∃ x ≠ ⊥ ∧ y ≠ ⊥)
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 · z', MkI · z) | z z'. d ≤ z' ∧ 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 (λz. nat (if z = ⊥ then d else (THE z'. z = MkI · 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 ≠ ⊥; y ≠ ⊥] =⇒ eq·x·y ≠ ⊥
begin

lemma eq-bottom-iff[simp]: (eq·x·y = ⊥) ↔ (x = ⊥ ∨ y = ⊥)
using eq-defined by auto
lemma \texttt{eq-defined-reflD[simp]}:
\begin{align*}
\text{(eq-a-a = TT)} & \iff a \neq \bot \\
\text{(TT = eq-a-a)} & \iff a \neq \bot \\
\text{a \neq \bot} & \implies \text{eq-a-a = TT}
\end{align*}
using \texttt{eq-refl} by \texttt{auto}

lemma \texttt{eq-FF[simp]}:
\begin{align*}
\text{(FF = eq-xs-ys)} & \iff (xs \neq \bot \land ys \neq \bot \land xs \neq ys) \\
\text{(eq-xs-ys = FF)} & \iff (xs \neq \bot \land ys \neq \bot \land xs \neq ys)
\end{align*}
by \texttt{(metis (mono-tags, hide-lams)} \texttt{Exh-tr dist-eq-tr(5) eq-TT-dest eq-bottom-iff eq-self-neq-FF)}+

lemma \texttt{eq-TT[simp]}:
\begin{align*}
\text{(TT = eq-xs-ys)} & \iff (xs \neq \bot \land ys \neq \bot \land xs = ys) \\
\text{(eq-xs-ys = TT)} & \iff (xs \neq \bot \land ys \neq \bot \land xs = ys)
\end{align*}
by \texttt{(auto simp: local.eq-TT-dest)}

end

instance \texttt{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
\texttt{CLetrec :: (\'a::pcpo \rightarrow \'a \times \'b::pcpo) \rightarrow \'b where}
\texttt{CLetrec = (\Lambda F. prod.sn (F::\mu x. prod.fst (F\cdot x)))}

nonterminal \texttt{recbinds and recbindt and recbind}

syntax
\texttt{-recbind :: logic \Rightarrow logic \Rightarrow recbind \ ((2- =/ -) 10) \ \Rightarrow recbind \Rightarrow recbindt \ (\cdot)}
\texttt{-recbindt :: recbind \Rightarrow recbindt \Rightarrow recbindt \ (-,/ -) \ \Rightarrow recbindt \Rightarrow recbinds \ (\cdot)}
\texttt{-recbinds :: recbindt \Rightarrow recbinds \Rightarrow recbinds \ (-;/ -) \ \Rightarrow recbinds \Rightarrow recbinds \ (\cdot\;/ \cdot)}
\texttt{-Letrec :: recbinds \Rightarrow logic \Rightarrow logic \ ((Letrec \cdot\cdot in \cdot) 10)}

translations
\begin{align*}
\texttt{(recbindt) x = a, (y,ys) = (b,bs) == (recbindt) (x,y,ys) = (a,b,bs)} \\
\texttt{(recbindt) x = a, y = b == (recbindt) (x,y) = (a,b)}
\end{align*}

translations
\begin{align*}
\texttt{-Letrec (-recbinds b bs) e == -Letrec b (-Letrec bs e)} \\
\texttt{Letrec x\cdot\cdot a in (e,es) == CONST CLetrec(\Lambda x\cdot\cdot (a,e,es))} \\
\texttt{Letrec x\cdot\cdot a in e == CONST CLetrec(\Lambda x\cdot\cdot (a,e))}
\end{align*}

3 Strict lists

Head- and tail-strict lists. Many technical Isabelle details are lifted from \texttt{HOLCF–Prelude.Data-List}; names follow HOL, prefixed with \texttt{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 (clarsimp 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 ꞇy 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 xx; 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:] ("[::<]")
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:] ("[::<]")
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
\[\left[ x \neq \perp; xs \neq \perp \right] \implies eq\text{-}slist\cdot(x :\# \; xs)\cdot[::] = \text{FF}\]
\[\left[ y \neq \perp; ys \neq \perp \right] \implies eq\text{-}slist\cdot[::] \cdot (y :\# \; ys) = \text{FF}\]
\[\left[ x \neq \perp; xs \neq \perp; y \neq \perp; ys \neq \perp \right] \implies eq\text{-}slist\cdot(x :\# \; xs)\cdot(y :\# \; ys) = (eq\cdot x\cdot y \text{ andalso } eq\text{-}slist\cdot xs\cdot ys)\]

instance proof

fix \(xs :: [':a]\)
show \(eq\cdot xs\cdot \perp = \perp\)
  by (cases \(xs\)) (subst eq\text{-}slist\cdot unfold; simp)+
show \(eq\cdot \perp \cdot xs = \perp\)
  by (cases \(xs\)) (subst eq\text{-}slist\cdot unfold; simp)+
qed

end

instance \(\text{slist} :: (Eq\text{-}sym)\) \(Eq\text{-}sym\)

proof
fix \(xs\; ys :: [':a]\)
show \(eq\cdot xs\cdot ys = eq\cdot ys\cdot xs\)
proof (induct \(xs\) arbitrary: \(ys\))
  case \(\text{snil}\)
    show \(?case\) by (cases \(ys\); simp)
  next
    case \(\text{scons}\)
    then show \(?case\) by (cases \(ys\); simp add: eq\text{-}sym)
qed simp\text{-}all

qed

instance \(\text{slist} :: (Eq\text{-}equiv)\) \(Eq\text{-}equiv\)

proof
fix \(xs\; ys\; zs :: [':a]\)
show \(eq\cdot xs\cdot zs \neq \text{FF}\)
  by (induct \(zs\)) simp\text{-}all
assume \(eq\cdot xs\cdot ys = \text{TT}\) and \(eq\cdot ys\cdot zs = \text{TT}\) then show \(eq\cdot xs\cdot zs = \text{TT}\)
proof (induct \(xs\) arbitrary: \(ys\; zs\))
  case \(\text{snil}\; ys\; zs\)
  then show \(?case\) by (cases \(ys\); simp\text{-}all)
next
  case \(\text{scons}\; x\; xs\; ys\; zs\)
  with eq\text{-}trans
  show \(?case\)
    by (cases \(ys\); cases \(zs\)) auto
qed simp\text{-}all

qed

instance \(\text{slist} :: (Eq\text{-eq})\) \(Eq\text{-eq}\)

proof
fix \(xs\; ys :: [':a]\)
show \(eq\cdot xs\cdot xs \neq \text{FF}\)
  by (induct \(xs\)) simp\text{-}all
assume \(eq\cdot xs\cdot ys = \text{TT}\) then show \(xs = ys\)
proof (induct \(xs\) arbitrary: \(ys\))
  case \(\text{snil}\; ys\)
  then show \(?case\) by (cases \(ys\)) simp\text{-}all
next
  case \(\text{scons}\; x\; xs\; ys\)
  then show \(?case\) by (cases \(ys\)) auto
qed simp

qed

instance \(\text{slist} :: (Eq\text{-def})\) \(Eq\text{-def}\)

proof
fix \(xs\; ys :: [':a]\)
assume \(xs \neq \perp\) and \(ys \neq \perp\)


then show \(eq::xs \cdot ys \neq \bot\)
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
definition sset :: ['a::Eq] 
where sset \(xs\) = \{x. slistmem x xs\}

3.1 Some of the usual reasoning infrastructure

inductive slistmem :: 'a ⇒ [:a::Eq] ⇒ bool where
\[\begin{array}{c}
[x \neq \bot; xs \neq \bot] \implies slistmem x (x :# xs) \\
[slistmem x xs; y \neq \bot] \implies slistmem x (y :# xs)
\end{array}\]

lemma slistmem-bottom1[iff]:
fixes \(x\) :: [:a::Eq]
shows \(- slistmem x \bot\)
by rule (induct \(x \bot::[:a:]\) rule: slistmem.induct; fastforce)

lemma slistmem-bottom2[iff]:
fixes \(xs\) :: [:a::Eq]
shows \(- slistmem \bot xs\)
by rule (induct \(\bot::[:a] xs\) rule: slistmem.induct; fastforce)

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

lemma slistmem-scons[simp]:
shows slistmem x (y :# ys) \iff (x = y ∧ x \neq \bot ∧ ys \neq \bot) ∨ (slistmem x ys ∧ y \neq \bot)
proof –
have \(x = y \lor slistmem x ys\) 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::] ⇒ 'a set where
\(sset xs = \{x. slistmem x xs\}\)

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

lemma sset-defined[iff]:
assumes \( x \in \text{sset } xs \)
shows \( x \neq \bot \)
using \text{assms sset-def} by \text{force}

\text{lemma sset-below:}
assumes \( y \in \text{sset } ys \)
assumes \( xs \subseteq ys \)
assumes \( xs \neq \bot \)
obtains \( x \) where \( x \in \text{sset } xs \) and \( x \subseteq y \)
using \text{assms}

\text{proof (induct } ys \text{ arbitrary: } xs) \quad \text{case } (\text{scons } y \text{ } ys \text{ } xs) \text{ then show } ?\text{case by (cases } xs) \text{ auto}
\text{qed simp-all}

3.2 Some of the usual operations

A variety of functions on lists. Drawn from Bird (1987), \textit{HOL.List} and \textit{HOLCF–Prelude.Data-List}. The definitions vary because, for instance, the strictness of some of those in \textit{HOLCF–Prelude.Data-List} correspond neither to those in Haskell nor Bird’s expectations (specifically \textit{stails, inits, ss.scanl}).

\text{fixrec snull :: [':a:] \to tr where}
\begin{align*}
\text{snull}[::] &= \text{TT} \\
| [x \neq \bot; xs \neq \bot] &\Longrightarrow \text{snull}(x : \# xs) = \text{FF}
\end{align*}

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

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

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

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

\text{lemma snull-eq-snul: } \text{snull} \cdot xs = \text{eq} \cdot xs[:]
\text{by (cases } xs) \text{ simp-all}

\text{fixrec smap :: (':a -> ':b) \to [':a:] \to [':b:] where}
\begin{align*}
\text{smap-f}[::] &= [::] \\
| [x \neq \bot; xs \neq \bot] &\Longrightarrow \text{smap-f}(x : \# xs) = f \cdot x : \# \text{smap-f} \cdot xs
\end{align*}

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

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

\text{lemma smap-is-snul-conv[simp]:}
\begin{align*}
(\text{smap-f} \cdot xs = [::]) &\longleftrightarrow (xs = [::]) \\
([:] = \text{smap-f} \cdot xs) &\longleftrightarrow (xs = [::])
\end{align*}
\text{by (cases } xs; \text{ simp)+}

\text{lemma smap-strict-scons[simp]:}
assumes \( f \cdot \bot = \bot \)
shows \( \text{smap-f}(x : \# xs) = f \cdot x : \# \text{smap-f} \cdot xs \)
using \text{assms by (cases } x : \# xs = \bot; \text{ fastforce)}
lemma \( \text{smap-ID'} \): \( \text{smap-ID} \cdot \text{xs} = \text{xs} \)
by (induct \text{xs}) simp-all

lemma \( \text{smap-ID[simp]} \): \( \text{smap-ID} = \text{ID} \)
by (clarsimp simp: cfun-eq-iff \text{smap-ID'})

lemma \( \text{smap-cong} \):
assumes \( \text{xs} = \text{xs}' \)
assumes \( \forall x. x \in \text{sset} \text{xs} \implies f \cdot x = f' \cdot x \)
shows \( \text{smap} f \cdot \text{xs} = \text{smap} f' \cdot \text{xs}' \)
using asms by (induct \text{xs} arbitrary: \text{xs}') auto

lemma \( \text{smap-smap}''[\text{simp}] \):
assumes \( f \cdot \bot = \bot \)
shows \( \text{smap} f \circ \text{smap} g \cdot \text{xs} = \text{smap} (f \circ g) \cdot \text{xs} \)
using asms by (clarsimp simp: cfun-eq-iff)

lemma \( \text{sset-smap}''[\text{simp}] \):
assumes \( \forall x. x \in \text{sset} \text{xs} \implies f \cdot x \neq \bot \)
shows \( \text{sset} (\text{smap} f \cdot \text{xs}) = \{ f \cdot x | x. x \in \text{sset} \text{xs} \} \)
using asms by (induct \text{xs}) auto

lemma \( \text{shedd-smap-distr}'' \):
assumes \( f \cdot \bot = \bot \)
assumes \( \forall x. x \in \text{sset} \text{xs} \implies f \cdot x \neq \bot \)
shows \( \text{shedd} (\text{smap} f \cdot \text{xs}) = f \circ \text{shedd} \cdot \text{xs} \)
using asms by (induct \text{xs}) simp-all

fixrec \( \text{sappend} :: \{ :: \} \rightarrow \{ :: \} \rightarrow \{ :: \} \) where
\( \text{sappend} \cdot [::] = [::] \) when \( \text{xs} :@ \text{ys} = \text{ys} \)
| \( \{ x \neq \bot; \text{xs} \neq \bot \} \implies \text{sappend} \cdot (x :# \text{xs}) \cdot \text{ys} = x :# \text{sappend} \cdot \text{xs} \cdot \text{ys} \)

abbreviation \( \text{sappend-syn} :: 'a \text{lslit} \Rightarrow 'a \text{lslit} \Rightarrow 'a \text{lslit} \) (infixr \( @ \) 65) where
\( \text{xs} :@ \text{ys} \equiv \text{sappend} \cdot \text{xs} \cdot \text{ys} \)

lemma \( \text{sappend-strict}'[\text{simp}] \): \( \text{sappend} \cdot \bot = \bot \)
by fixrec-simp

lemma \( \text{sappend-strict2}[\text{simp}] \): \( \text{xs} :@ \bot = \bot \)
by (induct \text{xs}) simp-all

lemma \( \text{sappend-bottom-iff}'[\text{simp}] \): \( \text{xs} :@ \text{ys} = \bot \) \( \iff \) \( \text{xs} = \bot \lor \text{ys} = \bot \)
by (induct \text{xs}) simp-all

lemma \( \text{sappend-scons}[\text{simp}] \): \( x :# \text{xs} :@ \text{ys} = x :# \text{xs} :@ \text{ys} \)
by (cases \( x :# \text{xs} = \bot \); fastforce)

lemma \( \text{sappend-assoc}[\text{simp}] \): \( \text{xs} :@ \text{ys} :@ \text{zs} = \text{xs} :@ (\text{ys} :@ \text{zs}) \)
by (induct \text{xs}) simp-all

lemma \( \text{sappend-snil-id-left}'[\text{simp}] \): \( \text{sappend} \cdot [::] = \text{ID} \)
by (simp add: cfun-eq-iff)
lemma \text{append-nil-right}[iff]: \text{xs} : @ [::] = \text{xs}

by (induct \text{xs}) simp-all

lemma \text{snil-append-iff}[iff]: \text{xs} : @ \text{ys} = [::] \iff \text{xs} = [::] \land \text{ys} = [::]

by (induct \text{xs}) simp-all

lemma \text{smap-sappend}[simp]: \text{smap} \cdot f \cdot (\text{xs} : @ \text{ys}) = \text{smap} \cdot f : \text{xs} = \text{smap} \cdot f : \text{ys}

by (induct \text{xs}; cases \text{ys} = \bot; simp)

lemma \text{stail-sappend}: \text{stail} \cdot (\text{xs} : @ \text{ys}) = \langle \text{case} \text{xs} \text{of} \text{[::]} \Rightarrow \text{stail} \cdot \text{ys} | z : \# \Rightarrow \text{zs} : @ \text{ys} \rangle

by (induct \text{xs}) simp-all

lemma \text{stail-append2}[simp]: \text{xs} \neq [::] \implies \text{stail} \cdot (\text{xs} : @ \text{ys}) = \text{stail} \cdot \text{xs} : @ \text{ys}

by (induct \text{xs}) simp-all

lemma \text{sfilter-scons-let}:

\begin{align*}
\text{lemma sfilter-scons-let[iff]}: & \text{xs} : @ [::] = \text{xs} \\
\text{by (induct \text{xs}) simp-all} & \\
\text{lemma snil-append-iff[iff]}: & \text{xs} : @ \text{ys} = [::] \iff \text{xs} = [::] \land \text{ys} = [::] \\
\text{by (induct \text{xs}) simp-all} & \\
\text{lemma smap-sappend}[simp]: & \text{smap} \cdot f \cdot (\text{xs} : @ \text{ys}) = \text{smap} \cdot f : \text{xs} = \text{smap} \cdot f : \text{ys} \\
\text{by (induct \text{xs}; cases \text{ys} = \bot; simp)} & \\
\text{lemma stail-sappend}: & \text{stail} \cdot (\text{xs} : @ \text{ys}) = \langle \text{case} \text{xs} \text{of} \text{[::]} \Rightarrow \text{stail} \cdot \text{ys} | z : \# \Rightarrow \text{zs} : @ \text{ys} \rangle \\
\text{by (induct \text{xs}) simp-all} & \\
\text{lemma stail-append2}[simp]: & \text{xs} \neq [::] \implies \text{stail} \cdot (\text{xs} : @ \text{ys}) = \text{stail} \cdot \text{xs} : @ \text{ys} \\
\text{by (induct \text{xs}) simp-all} & \\
\text{lemma sfilter-scons-let}: & \\
\end{align*}
assumes $p \bot = \bot$
shows $\text{sfilter}\cdot p\cdot (x :\# \, xs) = (\text{let } xs' = \text{sfilter}\cdot p\cdot xs \text{ in } \text{If } p\cdot x \text{ then } x :\# \, xs' \text{ else } xs')$

unfolding Let-def using assms by simp

lemma sfilter-sappend\[simp\]: $\text{sfilter}\cdot p\cdot (xs \@ \, ys) = \text{sfilter}\cdot p\cdot xs \@ \, \text{sfilter}\cdot p\cdot ys$
by (cases ys; clarsimp) (induct xs; fastforce simp: If2-def[symmetric] split: If2-splits)

lemma sfilter-const-FF\[simp\]:
  assumes $xs \neq \bot$
  shows $\text{sfilter}\cdot (\Lambda \, x. \, \text{FF})\cdot xs = [::]$
using assms by (induct xs) simp-all

lemma sfilter-const-FF-conv\[simp\]: $\text{(sfilter}\cdot (\Lambda \, x. \, \text{FF})\cdot xs = [::]) \iff (xs \neq \bot)$
by auto

lemma sfilter-const-TT\[simp\]: $\text{sfilter}\cdot (\Lambda \, x. \, \text{TT})\cdot xs = xs$
by (induct xs) simp-all

lemma sfilter-cong:
  assumes $xs = xs'$
  assumes $\bigwedge x. \, x \in \text{sset} \, xs \Rightarrow p\cdot x = p'\cdot x$
  shows $\text{sfilter}\cdot p\cdot xs = \text{sfilter}\cdot p'\cdot xs'$
using assms by (induct xs arbitrary; $xs')$ auto

lemma sfilter-snil-conv\[simp\]: $\text{sfilter}\cdot p\cdot [::] = \text{salt} \cdot (\text{neg oo } p)\cdot xs = \text{TT}$
by (induct xs; force simp: If2-def[symmetric] split: If2-splits)

lemma sfilter-sfilter': $\text{sfilter}\cdot p\cdot (\text{sfilter}\cdot q\cdot xs) = \text{sfilter}\cdot (\Lambda \, x. \, q\cdot x \text{ andalso } p\cdot x)\cdot xs$
proof (induct xs)
  case (scons $x \, xs$) from scons(1, 2) show ?case
    by (cases $\text{sfilter}\cdot q\cdot xs = \bot$)
      (simp-all add: If2-distr If2-andalso scons(3)[symmetric] del: sfilter-bottom-iff)
qed simp-all

lemma sfilter-sfilter: $\text{sfilter}\cdot p\cdot oo \, \text{sfilter}\cdot q = \text{sfilter}\cdot (\Lambda \, x. \, q\cdot x \text{ andalso } p\cdot x)$
by (clarsimp simp oo clarsimp simp sfilter-sfilter')

lemma sfilter-smap':
  assumes $p \bot = \bot$
  shows $\text{sfilter}\cdot p\cdot (\text{smap}\cdot f\cdot xs) = \text{smap}\cdot f\cdot (\text{sfilter}\cdot (p \, oo \, f)\cdot xs)$
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 $p \bot = \bot$
  shows $\text{sfilter}\cdot p\cdot oo \, \text{smap}\cdot f = \text{smap}\cdot f\cdot oo \, \text{sfilter}\cdot (p \, oo \, f)$
using assms by (clarsimp simp oo clarsimp simp sfilter-smap')

fixrec sfoldl :: \(['a::pcpo \rightarrow \, 'b::domain \rightarrow \, 'a) \rightarrow \, 'a \rightarrow \, [:\, 'b:]\rightarrow \, 'a\ where
sfoldl\cdot f\cdot [:\,] = z$
| \[x \neq \bot; \, xs \neq \bot\] \Rightarrow sfoldl\cdot f\cdot z\cdot (x :\# \, xs) = sfoldl\cdot f\cdot (f\cdot z\cdot x)\cdot xs

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

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

lemma sfoldl-cong:
  assumes xs = xs'
  assumes z = z'
  assumes ∃x z. x ∈ set xs → f·z·x = f'·z'·x
  shows sfoldl·f·z·xs = sfoldl·f'·z'·xs'
using assms by (cases xs arbitrary: xs' z z') auto

lemma sfoldl-sappend[simp]:
  assumes f·⊥ = ⊥
  shows sfoldl·f·z·(xs @ ys) = sfoldl·f·(sfoldl·f·z·xs)·ys
using assms by (cases ys = ⊥, force) (induct xs arbitrary: z; simp)

fixrec sfoldr :: ('a::pcpo → 'a) → 'a → ['a] → 'a where
  sfoldr·f·z·[::] = z
| [x ≠ ⊥; xs ≠ ⊥] → sfoldr·f·z·(x :: xs) = f·x·(sfoldr·f·z·xs)

lemma sfoldr-strict[simp]: sfoldr·f·z·⊥ = ⊥
by fixrec-simp

fixrec sconcat :: ['a:'] → ['a:'] where
  sconcat·[::] = [::]
| [x ≠ ⊥; xs ≠ ⊥] → sconcat·(x ::# xs) = x @ sconcat·xs

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

lemma sconcat-scons[simp]:
  shows sconcat·(x ::# xs) = x @ sconcat·xs
by (cases x = ⊥, force) (induct xs; fastforce)

lemma sconcat-sfoldl-aux: sfoldl·sappend·z·xs = z @ sconcat·xs
by (induct xs arbitrary: z) simp-all

lemma sconcat-sfoldl: sconcat = sfoldl·sappend·[::]
by (clarsimp simp: cfun-eq-iff sconcat-sfoldl-aux)

lemma sconcat-sappend[simp]: sconcat·(xs @ ys) = sconcat·xs @ sconcat·ys
by (induct xs) simp-all

fixrec slength :: ['a:] → Integer
where
  slength·[::] = 0
| [x ≠ ⊥; xs ≠ ⊥] → slength·(x ::# xs) = slength·xs + 1

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

lemma slength-bottom-iff[simp]: (slength·xs = ⊥) ↔ (xs = ⊥)
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
by (meson Integer.exhaust slength-bottom-iff slength-ge-0)
lemma srev-0-conv[simp]:
(slength-\(xs = 0\)) \iff (xs = [:::])
(slength-\(xs = MKI \cdot 0\)) \iff (xs = [:::])
eq 0 \cdot (slength-\(xs\)) = snull-\(xs\)
eq (slength-\(xs\)) \cdot 0 = snull-\(xs\)
by (induct xs) (auto simp: one-Integer-def elim: slengthE)

lemma le-length-0[simp]: (le 0 \cdot (slength-\(xs\)) = TT) \iff (xs \neq \bot)
by (cases slength-\(xs\)) (auto simp: slength-ge-0 zero-Integer-def)

lemma lt-length-0[simp]:
exs \neq \bot \implies lt (slength-\(xs\)) \cdot 0 = FF
exs \neq \bot \implies lt (slength-\(xs\)) \cdot (slength-\(xs\) + 1) = TT

unfolding zero-Integer-def one-Integer-def by (auto elim: slengthE)

lemma slength-smap[simp]:
assumes \(\forall x. x \neq \bot \implies f \cdot x \neq \bot\)
shows slength \cdot (smap f \cdot xs) = slength-\(xs\)
using assms by (induct xs) simp-all

lemma slength-sappend[simp]: slength \cdot (xs :@ ys) = slength-\(xs\) + slength-\(ys\)
by (cases ys = \bot, force) (induct xs; force simp: ac-simps)

lemma slength-sfoldl-\(\Lambda\) i \cdot i + 1 \cdot zs-\(zs\) = zs + slength-\(zs\)
by (induct zs arbitrary: z) (simp-all add: ac-simps)

lemma slength-sfoldl: slength = sfoldl \cdot (\Lambda i \cdot i + 1) \cdot 0
by (clarsimp simp: cfun-eq-iff slength-sfoldl-\(\Lambda\) i \cdot i + 1 \cdot 0)

lemma le-length-plus:
assumes \(xs \neq \bot\)
assumes \(n \neq \bot\)
shows le-n \cdot (slength-\(xs\) + \(n\)) = TT
using assms by (cases n; force elim: slengthE)

fixrec srev :: [\:'a:] \rightarrow [\:'a:] where
srev[\:'i:] = [\:'i:]
| \(x \neq \bot; \ \(xs \neq \bot\)\) \implies srev \cdot (x :@ xs) = srev-\(xs\) :@ [\:'x:]

lemma srev-strict[simp]: srev-\bot = \bot
by fixrec-simp

lemma srev-bottom-iff[simp]: (srev \cdot xs = \bot) \iff (xs = \bot)
by (induct xs) simp-all

lemma srev-scons[simp]: srev \cdot (x :@ xs) = srev-\(xs\) :@ [\:'x:]
by (cases x = \bot, clarsimp) (induct xs; force)

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

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

lemma srev-cases[case-names bottom snil ssnoc]:
assumes \(xs = \bot \implies P\)
assumes \(xs = [:::] \implies P\)
assumes $\forall y \ y \neq \bot; y \neq \bot; x = \{y\} \Rightarrow P$
shows $P$
using $\text{assms}$ by (metis list.exhaust srev.simps(1) srev-scons srev-srev-ident srev-strict)

**lemma** srev-induct[case-names bottom snil ssoc]:
assumes $P \bot$
assumes $P ::[$
assumes $\forall x \ x \neq \bot; x \neq \bot; P x \Rightarrow P (x :: [x:])$
shows $P x$
proof
have $P (\text{srev}(\text{srev}-\text{xs}))$ by (rule list.induct[where $x=\text{srev-}x$]; simp add: $\text{assms}$)
thен show $\text{?thesis}$ by simp
qed

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

**fixrec** stake :: Integer $\rightarrow$ $[::']$ $\rightarrow$ $[::']$ where — Note: strict in both parameters.

$\text{stake}\cdot \bot\cdot \bot = \bot$
| $i \neq \bot \Rightarrow \text{stake}\cdot i\cdot [] = []$
| $[[x \neq \bot; x \neq \bot]] \Rightarrow \text{stake}\cdot i\cdot (x :: x)$

**lemma** stake-strict[simp]:

$\text{stake}\cdot \bot = \bot$

$\text{stake}\cdot i\cdot \bot = \bot$

by fixrec-simp+

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

**lemma** stake-0[simp]:

$\text{stake}\cdot \bot\cdot x = []$
$\text{stake}\cdot i\cdot x = []$
$\text{stake}\cdot (\text{MkI}\cdot0)\cdot x = []$

by (cases $x$; simp add: zero-Integer-def)+

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

**lemma** take-MkI-scons[simp]:

$\forall n \Rightarrow \text{take}\cdot (\text{MkI}\cdot n)\cdot (x :: x) = x :: \text{take}\cdot (\text{MkI}\cdot (n - 1))\cdot x$
by (cases $x = \bot$; cases $x = \bot$; simp add: zero-Integer-def one-Integer-def)

**lemma** stake-numeral-scons[simp]:

$\forall x \neq \bot \Rightarrow \text{stake}\cdot i\cdot (x :: x) = []$

$\text{stake}\cdot (\text{numeral} \cdot \text{Num.Bit0} k)\cdot (x :: x) = x :: \text{stake}\cdot (\text{numeral} \cdot \text{Num.BitM} k)\cdot x$

$\text{stake}\cdot (\text{numeral} \cdot \text{Num.Bit1} k)\cdot (x :: x) = x :: \text{stake}\cdot (\text{numeral} \cdot \text{Num.Bit0} k)\cdot x$

by (cases $x = \bot$; cases $x$; simp add: zero-Integer-def one-Integer-def numeral-Integer-eq)+

**lemma** stake-all:
assumes le-slen-while-x$i = $TT
shows $\text{stake}\cdot i\cdot x = x$
using $\text{assms}$
proof(induct $x$ arbitrary; $i$)
case $\text{scons} x x i$ then show $\text{?case}$
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; simp add: zero-Integer-def)

next
proof
stake-append
lemma (stake-all-triv)
using lemma stake-all-triv simp add: zero-Integer-def
qed

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; simp add: zero-Integer-def)

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

lemma sdrop-snill-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 ≠ ⊥ ⇒ sdrop·I·(x :# xs) = xs
x ≠ ⊥ ⇒ sdrop·(numeral (Num.Bit0 k))·(x :# xs) = sdrop·(numeral (Num.BitM k))·xs
x ≠ ⊥ ⇒ 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)+

18
lemma \text{sdrop-sappend}[simp]:
\[ \text{sdrop}-i(\text{xs} \mathbin{@} \text{ys}) = \text{sdrop}-i(\text{xs}) \mathbin{@} \text{sdrop}-(i - \text{slength} \cdot \text{xs}) \cdot \text{ys} \]

proof (induct \text{xs} arbitrary: \( i \))
  case (\text{snil} \( i \)) then show ?case by (cases \( i \); simp add: zero-Integer-def)
next
  case (\text{scons} \( \text{x} \) \( \text{xs} \) \( i \)) then show ?case
    by (cases \( \text{ys} \) = \( \bot \); cases \( \text{le} \cdot 0 \); cases \( i \);
        clarsimp simp: zero-Integer-def one-Integer-def \text{sdrop}-neg \text{sdrop}-pos add commute diff diff add
        split: \text{if}-splits elim!: \text{slengthE})
qed simp

lemma \text{slength-sdrop}[simp]:
\[ \text{slength} - (\text{sdrop} \cdot \text{xs}) = \text{If} \ \text{le} \cdot 0 \ \text{then} \ \text{slength} - \text{xs} \ \text{else} \ \text{If} \ \text{le} - (\text{slength} - \text{xs}) - \text{i} \ \text{then} \ 0 \ \text{else} \ \text{slength} - \text{xs} - \text{i} \]

proof (induct \text{xs} arbitrary: \( i \))
  case (\text{snil} \( i \)) then show ?case by (cases \( i \); simp add: zero-Integer-def)
next
  case (\text{scons} \( \text{x} \) \( \text{xs} \) \( i \)) then show ?case
    by (subst \text{sdrop}.\text{unfold}; cases \( i \);
        clarsimp simp: If2-def[symmetric] zero-Integer-def one-Integer-def \text{sdrop}-neg \text{sdrop}-pos add commute diff diff add
        split: \text{if}-splits elim!: \text{slengthE})
qed (simp-all add: le-defined)

lemma \text{sdrop-not-snilD}:
assumes \text{sdrop} - (\text{MkI} \cdot \text{xs}) \neq [:]
assumes \( \text{xs} \neq \bot \)
shows \( \text{lt} \cdot (\text{MkI} \cdot \text{xs}) \cdot (\text{slength} - \text{xs}) = \text{TT} \land \text{xs} \neq [:] \)

using assms
proof (induct \text{xs} arbitrary: \( i \))
  case (\text{scons} \( \text{x} \) \( \text{xs} \) \( i \)) then show ?case
    by (subst (asm) (2) \text{sdrop}.\text{unfold}; clarsimp simp: zero-Integer-def one-Integer-def elim!: \text{slengthE})
qed simp-all

lemma \text{sdrop-sappend-same}:
assumes \( \text{xs} \neq \bot \)
shows \( \text{sdrop} - (\text{slength} \cdot \text{xs}) \cdot (\text{xs} \mathbin{@} \text{ys}) = \text{ys} \)

using assms
proof (induct \text{xs} arbitrary: \( \text{ys} \))
  case (\text{scons} \( \text{x} \) \( \text{xs} \) \( \text{ys} \)) then show ?case
    by (cases \( \text{ys} \) = \( \bot \); subst \text{sdrop}.\text{unfold}; clarsimp simp: zero-Integer-def one-Integer-def elim!: \text{slengthE})
qed simp-all

fixrec \text{sscanl} :: \( \text{('a} \to \text{'b} \to \text{'a}) \to \text{'a} \to [:'b:] \to [:'a:] \) where
\[ \text{sscanl} \cdot \text{f} \cdot z \cdot [:] = z \cdot :: [:] \]
| \( [x \neq \bot; \text{xs} \neq \bot] \Rightarrow \text{sscanl} \cdot \text{f} \cdot z \cdot (\text{x} \cdot :: \text{xs}) = z \cdot :: \text{sscanl} \cdot (\text{f} \cdot z \cdot x) \cdot \text{xs} \]

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

lemma sscanl-cong:
  assumes \( xs = xs' \)
  assumes \( z = z' \)
  assumes \( \forall x \ z \cdot x \in sset xs \Rightarrow f \cdot z \cdot x = f' \cdot z' \cdot x \)
  shows \( sscanl \cdot f \cdot z \cdot xs = sscanl \cdot f' \cdot z' \cdot xs' \)
using \texttt{assms by (induct \( xs \) arbitrary; \( xs' \ z \ z' \)) auto}

lemma sscanl-lfp-fusion':
  assumes \( g \cdot \bot = \bot \)
  assumes \( \ast \cdot \forall acc \ x \cdot x \neq \bot \Rightarrow g \cdot (f \cdot acc \cdot x) = f' \cdot (g \cdot acc) \cdot x \)
  shows \( smap \cdot g \cdot (sscanl \cdot f \cdot z \cdot xs) = sscanl \cdot f' \cdot (g \cdot z) \cdot xs \)
using \texttt{assms by (induct \( xs \) arbitrary; \( z \)) simp-all}

lemma sscanl-lfp-fusion:
  assumes \( g \cdot \bot = \bot \)
  assumes \( \ast \cdot \forall acc \ x \cdot x \neq \bot \Rightarrow g \cdot (f \cdot acc \cdot x) = f' \cdot (g \cdot acc) \cdot x \)
  shows \( smap \cdot g \circ oo \ sscanl \cdot f \cdot z = sscanl \cdot f' \cdot (g \cdot z) \cdot xs \)
using \texttt{assms by (clarsimp simp: cfun-eq-iff sscanl-lfp-fusion')}

lemma sscanl-ww-fusion': — Worker/wrapper (Gammie 2011; Gill and Hutton 2009) specialised to \texttt{sscanl}
fixes \texttt{wrap :: 'b \rightarrow 'a}
fixes \texttt{unwrap :: 'a \rightarrow 'b}
fixes \texttt{z :: 'a}
fixes \texttt{f :: 'a \rightarrow 'c \rightarrow 'a}
fixes \texttt{f' :: 'b \rightarrow 'c \rightarrow 'b}
assumes \texttt{ww: \forall \forall acc \cdot (acc \neq \bot \Rightarrow \texttt{unwrap f (wrap z) x = f' (unwrap (wrap z)) x}}
shows \( sscanl \cdot f \cdot z \cdot xs = \texttt{smap \cdot wrap oo sscanl \cdot f' \cdot (unwrap z) \cdot xs} \)
using \texttt{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 \texttt{sscanl}
fixes \texttt{wrap :: 'b \rightarrow 'a}
fixes \texttt{unwrap :: 'a \rightarrow 'b}
fixes \texttt{z :: 'a}
fixes \texttt{f :: 'a \rightarrow 'c \rightarrow 'a}
fixes \texttt{f' :: 'b \rightarrow 'c \rightarrow 'b}
assumes \texttt{ww: \forall \forall acc \cdot (acc \neq \bot \Rightarrow \texttt{unwrap f (wrap z) x = f' (unwrap (wrap z)) x}}
shows \( sscanl \cdot f \cdot z = \texttt{smap \cdot wrap oo sscanl \cdot f' \cdot (unwrap z)} \)
using \texttt{assms by (clarsimp simp: cfun-eq-iff sscanl-ww-fusion')}

fixrec \texttt{sinits : ['a:] \rightarrow [':a:]} where
\texttt{sinits :[]= [] :: [: []]} \texttt{| [x \neq \bot ; xs \neq \bot] \Rightarrow sinits (x ::# xs) = [] :: smap (scons x) (sinits xs)}

lemma \texttt{sinits-strict[simp]: sinits \bot = \bot}
by \texttt{fixrec-simp}

lemma \texttt{sinits-bottom-iff[simp]: (sinits xs = \bot) \leftrightarrow (xs = \bot)}
by (induct \( xs \)) simp-all

lemma \texttt{sinits-not-nil[iff]: sinits xs \neq []}
by (cases \( xs \)) simp-all

lemma \texttt{sinits-empty-bottom[simp]: (sset (sinits xs) = {}) \leftrightarrow (xs = \bot)}
by (cases \( xs \)) simp-all
lemma \textit{sinits-scons}[simp]: \(\text{sinits} \cdot (x :\# xs) = [::] :\# \text{smap} \cdot (x :\#)(\text{sinits} \cdot xs)\)
by (cases \(x = \bot\), force) (induct \(xs\); force)

lemma \textit{sinits-length}[simp]: \(\text{slength} \cdot (\text{sinits} \cdot xs) = \text{slength} \cdot xs + 1\)
by (induct \(xs\)) simp-all

lemma \textit{sinits-snoc}[simp]: \(\text{sinits} \cdot (xs :@ [::]) = \text{sinits} \cdot xs :@ [::] :@ [::] :::\)
by (induct \(xs\)) simp-all

lemma \textit{sinits-sfoldl}': — Bird (1987, p30)
shows \(\text{sinits} \cdot xs = \text{sfoldl} \cdot (\Lambda \ x \ ys. \ [::] :@ \text{smap} \cdot (x :\#) \cdot ys) \cdot [::] :@ [::] :::\) \(\cdot xs\)
by (induct \(xs\)) simp-all

lemma \textit{sinits-sscanl}': — Bird (1987, Lemma 5), Bird (2010, p118 “the scan lemma”)
shows \(\text{smap} \cdot (\text{sfoldl} \cdot f \cdot z) \cdot (\text{sinits} \cdot xs) = \text{sscanl} \cdot f \cdot z \cdot xs\)
by (simp add: \textit{sinits-sscanl}' cfun-eq-iff)

lemma \textit{sinits-all}[simp]: \((xs \in \text{sset} \cdot (\text{sinits} \cdot xs)) \iff (xs \neq \bot)\)
by (induct \(xs\)) simp-all

fixrec \textit{stails} :: [':a:] \rightarrow [::] [':a:]. where
\begin{itemize}
  \item \(\text{stails} :: [::] = [::] :\# [::]\)
  \item \(\lfloor x \neq \bot; xs \neq \bot \rfloor \Rightarrow \text{stails} \cdot (x :\# xs) = (x :\# xs) :\# \text{stails} \cdot xs\)
\end{itemize}

lemma \textit{stails-strict}[simp]: \(\text{stails} \cdot \bot = \bot\)
by fixrec-simp

lemma \textit{stails-bottom-iff}[simp]: \((\text{stails} \cdot xs = \bot) \iff (xs = \bot)\)
by (induct \(xs\)) simp-all

lemma \textit{stails-not-snil}[iff]: \(\text{stails} \cdot xs \neq [::]\)
by (cases \(xs\)) simp-all

lemma \textit{stails-scons}[simp]: \(\text{stails} \cdot (x :\# xs) = (x :\# xs) :\# \text{stails} \cdot xs\)
by (induct \(xs\)) (cases \(x = \bot\); simp)+

lemma \textit{stails-slength}[simp]: \(\text{slength} \cdot (\text{stails} \cdot xs) = \text{slength} \cdot xs + 1\)
by (induct \(xs\)) simp-all

lemma \textit{stails-snoc}[simp]:
shows \(\text{stails} \cdot (xs :@ [::]) = \text{smap} \cdot (\Lambda \ ys. \ [::] :@ \text{stails} \cdot ys) :@ [::] :::\)
by (induct \(xs\)) simp-all

lemma \textit{stails-sfoldl}':
shows \(\text{stails} \cdot xs = \text{sfoldl} \cdot (\Lambda \ x \ ys. \ [::] :@ \text{smap} \cdot (x :\#) \cdot ys) \cdot [::] :@ [::] :::\) \(\cdot xs\)
by (induct \(xs\) rule: \textit{srev-induct}) simp-all

lemma \textit{stails-sfoldl}:
shows \(\text{stails} = \text{sfoldl} \cdot (\Lambda \ x \ ys. \ [::] :@ \text{smap} \cdot (x :\#) \cdot ys) \cdot [::] :@ [::] :::\)
by (clarsimp simp: cfun-eq-iff \textit{stails-sfoldl}')

lemma \textit{stails-all}[simp]: \((xs \in \text{sset} \cdot (\text{stails} \cdot xs)) \iff (xs \neq \bot)\)
by (cases \(xs\)) simp-all

21
\textbf{fixrec} \texttt{sconcatMap} :: \texttt{('}a\texttt{→}\texttt{[}:'a:\texttt{]}) \rightarrow \texttt{[}:'a:\texttt{]→} \texttt{[}:'a:\texttt{]} \where\n\texttt{[simp def]}: \texttt{sconcatMap.f} = \texttt{sconcat o smap.f}

\textbf{lemma} \texttt{sconcatMap.strict[simp]}: \texttt{sconcatMap.f} \cdot \bot = \bot
\textbf{by} \texttt{fixrec-simp}

\textbf{lemma} \texttt{sconcatMap-snill[simp]}: \texttt{sconcatMap.f} \cdot \texttt{[]} = \texttt{[]}
\textbf{by} \texttt{fixrec-simp}

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

\textbf{lemma} \texttt{sconcatMap-bottom-iff[simp]}: (\texttt{sconcatMap.f} \cdot \texttt{xs} = \bot) \leftrightarrow (\texttt{xs} = \bot \lor (\exists \texttt{x}\in\texttt{sset} \texttt{xs}. \texttt{f} \cdot \texttt{x} = \bot))
\textbf{by} \texttt{(induct} \texttt{xs} \texttt{) simp-all}

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

\textbf{lemma} \texttt{sconcatMap-monad-laws:}
\begin{align*}
\texttt{sconcatMap}(\Lambda \texttt{x}. \texttt{[}:'x:\texttt{]}\cdot \texttt{xs}) &= \texttt{xs} \\
\texttt{sconcatMap.g}(\texttt{sconcatMap.f} \cdot \texttt{xs}) &= \texttt{sconcatMap}(\Lambda \texttt{x}. \texttt{sconcatMap.g} \cdot (\texttt{f} \cdot \texttt{x})) \cdot \texttt{xs}
\end{align*}
\textbf{by} \texttt{(induct} \texttt{xs} \texttt{) simp-all}

\textbf{fixrec} \texttt{supto} :: \texttt{Integer} \rightarrow \texttt{Integer} \rightarrow \texttt{[}:'\texttt{Integer}\texttt{]} \where
\texttt{[simp del]}: \texttt{supto.i-j} = \texttt{If \texttt{le.i-j} then \texttt{i} :\# \texttt{supto.(i+1).j} else \texttt{[]}}

\textbf{lemma} \texttt{upto-strict[simp]}:
\begin{align*}
\texttt{supto.}\bot &= \bot \\
\texttt{supto.m.}\bot &= \bot
\end{align*}
\textbf{by} \texttt{fixrec-simp}
proof (simp):

lemma supto-snil-conv [simp]:
(supto(MkI·i) · MkI·j) = ::)  (j < i)
(induction i arbitrary: 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 = ::;

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: 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·j) · MkI·j-1)

proof (induct (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: 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 \( j - i = - \text{ int } d \)
thен have \( j \leq i \) by auto
moreover
\{ assume \( j = i \) then have \( \forall P i j \) by (simp add: one-Integer-def) \}
moreover
\{ assume \( j < i \) then have \( \forall P i j \) by (simp add: one-Integer-def) \}
ultimately show \( \text{thesis by arith} \)
qed

lemma \( \text{sset-supto[simp]}: \)
\( \text{sset (\text{supto-}(\text{MkI} \cdot i)\text{-}(\text{MkI} \cdot j))} = \{ \text{MkI} \cdot k \mid k. \ i \leq k \land k \leq j \} \) (is \( \text{sset (\forall u i j) = \text{?R i j}} \)
proof (cases \( j - i \))
case (nonneg \( k \))
then show \( \text{thesis} \)
proof (induct \( k \) arbitrary: \( i j \))
case (Suc \( k \))
then have \( \ast: j - (i + 1) = \text{int } k \) by simp
from \( \text{Suc(1)}[\text{OF } \ast] \) have \( \text{IH: sset (\forall u (i+1) j) = \text{?R (i+1) j}} \).
from \( \ast \) have \( i \leq j \) by simp
then have \( \text{sset (\forall u i j) = \text{sset (MkI} \cdot i :\# \text{ ?u (i+1) j}} \) by (simp add: one-Integer-def)
also have \( \ldots = \text{insert (MkI} \cdot i) (\text{?R (i+1) j}} \) by (simp add: \( \text{IH} \))
also have \( \ldots = \text{?R i j using i \leq j \ by auto} \)
finally show \( \text{?case} \).
qed (force simp: one-Integer-def)
qed simp

lemma \( \text{supto-split1: } \) — From HOL.List
assumes \( i \leq j \)
assumes \( j \leq k \)
shows \( \text{supto-}(\text{MkI} \cdot i)\text{-}(\text{MkI} \cdot k) = \text{supto-}(\text{MkI} \cdot i)\text{-}(\text{MkI} \cdot (j - 1)) :\@ \text{supto-}(\text{MkI} \cdot j)\text{-}(\text{MkI} \cdot k) \)
using assms
proof (induct \( j \) rule: int_ge_induct)
case (step \( l \)) with \( \text{supto-simp(2)} \) \text{supto-snoc } \text{?case by (clarsimp simp: one-Integer-def)}
qed simp

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

lemma \( \text{supto-split3: } \) — From HOL.List
assumes \( i \leq j \)
assumes \( j \leq k \)
shows \( \text{supto-}(\text{MkI} \cdot i)\text{-}(\text{MkI} \cdot k) = \text{supto-}(\text{MkI} \cdot i)\text{-}(\text{MkI} \cdot (j - 1)) :\@ \text{ MkI} \cdot j :\# \text{supto-}(\text{MkI} \cdot (j + 1))\text{-}(\text{MkI} \cdot k) \)
using assms \text{supto-simp(2)} \text{supto-split1 by (metis one-Integer-def plus-MkI-MkI) }

lemma \( \text{sinits-stake‘: } \)
shows \( \text{sinits-zs = smap}(\Lambda \ i. \ \text{stake-i-zs})\text{-}(\text{supto-}(\text{length-} \text{xs})) \)
proof(induct \( \text{xs} \) rule: srev-induct)
case (ssnoc \( x \) \( \text{xs} \) ) then show \( \text{?case} \)
apply (clarsimp simp: zero-Integer-def one-Integer-def stake-all
  simp del: supto-simp
  elim!: lengthE)
apply (rule arg-cong, rule smap-cong[of refl])
apply clarsimp
done
qed simp-all

lemma tails-sdrop':
shows tails-xs = smap (Π 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 tail-sdrop
    simp del: supto-simp
    elim!: lengthE)
  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 ≠ ⊥
shows sdrop (MkI i xs) ∈ sset (stails xs)
using assms by (clarsimp simp: stails-sdrop zero-Integer-def elim!: lengthE)
  (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 ≠ ⊥
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 xs) 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)[ys] = 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 (snl 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 x [:] = (case xs of [:] ⇒ TT | x :# xs ⇒ FF)
bysimp

lemma prefix-singleton-TT:
  shows prefix [:] ys = TT ↔ (x ⊥ ∧ (∃ zs. zs ⊥ ∧ ys = x :# zs))
bysimp cases xs; fastforce

lemma prefix-singleton-FF:
  shows prefix [:] ys = FF ↔ (x ⊥ ∧ (ys = [:] ∨ (∃ z zs. z ⊥ ∧ zs ⊥ ∧ ys = z :# zs ∧ x = z)))
bysimp cases xs; 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 (snl 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)
lemma prefix-slength-strengthen: \( \text{prefix-}xs\cdot ys = (\text{le-}(\text{slength-}xs)\cdot (\text{slength-}ys) \text{ andalso prefix-}xs\cdot ys) \)

by (rule andalso-weaken-left) (auto dest: prefix-slength)

lemma prefix-scons-snil[simp]: \( \text{prefix-}(x :\# xs)\cdot [:] \neq TT \)

by (cases \( x :\# xs \neq \bot \)) auto

lemma scons-prefix-scons[simp]:

\[
(\text{prefix-}(x :\# xs)\cdot (y :\# ys) = TT) \iff (eq\cdot x\cdot y = TT \land \text{prefix-}xs\cdot ys = TT)
\]

by (cases \( x :\# xs \neq \bot \land y :\# ys \neq \bot \)) auto

lemma append-prefixD:

assumes \( \text{prefix-}(xs :@ ys)\cdot zs = TT \)

shows \( \text{prefix-}xs\cdot zs = TT \)

using assms

proof (induct \( xs \) arbitrary: zs)

  case (snil zs) then show \( \text{thesis} \) by (simp)

  next

    case (scons \( x \) \( xs \) zs) then show \( \text{thesis} \) by (simp)

      by (metis prefix.simps(1) prefix-scons-snil sappend-scons scons-prefix-scons slist.exhaust)

qed simp

lemma same-prefix-prefix[simp]:

assumes \( xs \neq \bot \)

shows \( \text{prefix-}(xs :@ ys)\cdot (xs :@ zs) = \text{prefix-}ys\cdot zs \)

using assms

proof (cases \( ys \) arbitrary: zs)

  case False-False with assms show \( \text{thesis} \) by (simp)

qed simp-all

lemma eq-prefix-TT:

assumes \( eq\cdot xs\cdot ys = TT \)

shows \( \text{prefix-}xs\cdot ys = TT \)

using assms by (induct \( xs \) arbitrary: \( ys \)) (case-tac simp; simp)+

lemma prefix-eq-FF:

assumes \( eq\cdot xs\cdot ys = FF \)

shows \( eq\cdot xs\cdot ys = FF \)

using assms by (induct \( xs \) arbitrary: \( ys \)) (case-tac auto)+

lemma prefix-slength-eq:

shows \( eq\cdot xs\cdot ys = (eq\cdot (\text{slength-}xs)\cdot (\text{slength-}ys) \text{ andalso prefix-}xs\cdot ys) \)

proof (induct \( xs \) arbitrary: \( ys \))

  case (snil \( ys \)) then show \( \text{thesis} \)

    by (cases \( ys \); clarsimp simp: one-Integer-def elim!: slengthE)

  next

    case (scons \( x \) \( xs \) \( ys \)) then show \( \text{thesis} \)

      by (cases \( ys \); clarsimp simp: zero-Integer-def one-Integer-def elim!: slengthE)

qed simp

lemma stake-slength-plus-1:

shows \( \text{stake-}(\text{slength-}xs + 1)\cdot (y :\# ys) = y :\# \text{ stake-}(\text{slength-}xs)\cdot ys \)

by (cases \( xs = \bot \); \( y = \bot \); \( ys = \bot \)) rule: bool.exhaust [case-product bool.exhaust bool.exhaust]; clarsimp

(auto simp: If2-def[rule: symmetric] zero-Integer-def one-Integer-def elim!: slengthE)

lemma sdrop-slength-plus-1:

assumes \( y \neq \bot \)
shows \( \text{sdrop}(\text{slength} \cdot \text{xs} + 1) \cdot (y : \# \text{ys}) = \text{sdrop}(\text{slength} \cdot \text{xs}) \cdot \text{ys} \)

using assms
by (subst \text{sdrop} . \text{simps} ;
\begin{itemize}
  \item cases \text{xs} = \bot ; \text{clarsimp} ; \text{cases \text{ys} = \bot ;}
  \item \text{clarsimp simp: If2-def[symmetric] zero-Integer-def one-Integer-def split: If2-splits elim!: slengthE}
\end{itemize}

lemma \( \text{eq-take-length-prefix} \): \( \text{prefix} \cdot \text{xs} \cdot \text{ys} = \text{eq} \cdot \text{xs} \cdot (\text{stake} \cdot (\text{slength} \cdot \text{xs}) \cdot \text{ys}) \)

proof (induct \text{xs arbitrary: \text{ys}})
\begin{itemize}
  \item case (\text{snil \text{ys}}) show \( \text{thesis by simp} \)
  \item case (\text{scons \text{x \text{xs} \text{ys}}})
  \begin{itemize}
    \item note \( \text{IH} = \text{this} \)
    \item show \( \text{thesis by simp} \)
  \end{itemize}
\end{itemize}

The following examples illustrate this behaviour:

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 \( \text{eq} \) has some nice properties (see §2.2) and use strict lists.

fixrec \text{endswith} :: [:\text{\'a::Eq-def}] \rightarrow [:\text{\'a}] \rightarrow \text{tr}
where
\begin{itemize}
  \item [simp def]: \text{endswith-pat} = \text{selem-pat oo stails}
\end{itemize}

fixrec \text{matches} :: [:\text{\'a::Eq-def}] \rightarrow [:\text{\'a}] \rightarrow [:\text{Integer}]
where
\begin{itemize}
  \item [simp def]: \text{matches-pat} = \text{smap-slengh oo } \text{sfilt} (\text{endswith-pat}) \text{ oo sinits}
\end{itemize}

Bird describes \( \text{matches-pat} \cdot \text{xs} \) as returning “a list of integers \( p \) such that \( \text{pat} \) is a suffix of \( \text{stake} \cdot \text{p} \cdot \text{xs} \).”

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, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3, 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]:
  \(\text{endswith} \cdot \bot = \bot\)
  \(\text{endswith-pat} \cdot \bot = \bot\)
by \((\text{fixrec-simp}; simp add: cfun-eq-iff)\)+

lemma matches-strict[simp]:
  \(\text{matches} \cdot \bot = \bot\)
  \(\text{matches-pat} \cdot \bot = \bot\)
by \((\text{fixrec-simp}; clarsimp simp: cfun-eq-iff)\)+

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-(\(\Lambda\) (\(a, b\) \(z\). \((f1\text{-}\text{a-z}, f2\text{-}\text{b-z})\))\)(\(z1\), \(z2\)) \((\text{is } \text{?lhs = ?rhs)}\)
proof \((\text{rule cfun-eqI})\)
  fix \(xs\) show \(?lhs\cdot\text{xs} = \text{?rhs}•\text{xs}\)
    by \((\text{induct xs arbitrary: z1 z2})\) simp-all
qed

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

lemma Bird-strategy:
  assumes endswith: endswith-pat = \(p \text{ oo } \text{sfoldl-op-z}\)
  assumes step: step = \(\Lambda\) (\(n, x\) \(y\). \((n + 1, \text{op-x} • y))\)
  assumes \(p, \bot = \bot\) — We can reasonably expect the predicate to be strict
  shows matches-pat = \(\text{smap-} \text{cfst oo } \text{sfilter-} \cdot (\text{p oo } \text{csnd}) \text{ oo } \text{sscanl-step} \cdot (0, z)\)
apply \((\text{simp add: matches.simps assoc-oo endswith})\)
apply \((\text{subst map-}\text{sfilter-}\text{split-cfcomp, fastforce, fact})\)
apply \((\text{subth length-}\text{sfoldl})\)
apply \((\text{subth fork-}\text{sfoldl})\)
apply \((\text{simp add: oo-assoc[ symmetric]}))\)
apply \((\text{subth sinitis-sscanl})\)
apply \((\text{fold step})\)
apply \((\text{rule refl})\)
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 \(0 \leq d\)
  shows \(\text{sfilter-} \cdot (\Lambda \text{x. le-} (\text{MkI} • n - \text{x}) • (\text{MkI} • d)) \cdot (\text{supto-} (\text{MkI} • m) • (\text{MkI} • n))\)
proof(cases m ≤ n − d)
  case True
  moreover from True assms have ?filterp · ?suptomn = ?filterp · (supto · (MkI · m) · (MkI · (n − d − 1))) ⋁ (supto · (MkI · (n − d − 1)) · (MkI · n))
    using supto-split1 by auto
  moreover from True assms have ?filterp · (supto · (MkI · m) · (MkI · (n − d − 1))) = [:] by auto
next
  case False then show ?thesis
    by (clarsimp intro: trans[OF filter-cong[OF refl] filter-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 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 · ((filter · (Λ x · prefix · x) · (stails · xs)))) · (is ?lhs = ?rhs)
proof(cases pat = ⊥ xs = ⊥ rule: bool.exhaust[case-product bool.exhaust])
  case False-
  from 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 ≤ xs then xsl − patl else 0
  have ?rhs = eq-pat · (shead · ((filter · (Λ x · le · (slength · x) · (slength · pat) andalso prefix · x) · (stails · xs)))
    by (subst prefix-length-strengthen) simp
  also have . . . = eq-pat · (shead · (filter · (Λ x · prefix · x) · (filter · (Λ x · le · (slength · x) · (slength · pat) · (stails · xs)))
    by (simp add: filter-filter')
  also have . . . = eq-pat · (shead · (smap · (Λ k · sdrop · k · xs) · (filter · (Λ k · prefix · (sdrop · k · xs) · pat) · (filter · (Λ k · le · (slength · (sdrop · k · xs) − k) · (MkI · patl)) · (supto · (MkI · 0) · (MkI · xsl)))))
    using (slength · xs = MkI · xsl)
    by (simp add: stails-sdrop' filter-smap' cfcomp1 zero-Integer-def)
  also have . . . = eq-pat · (shead · (smap · (Λ k · sdrop · k · xs) · (filter · (Λ k · prefix · (sdrop · k · xs) · pat) · (filter · (Λ k · le · (MkI · xsl − k) · (MkI · patl)) · (supto · (MkI · 0) · (MkI · xsl))))))
  using (slength · xs = MkI · xsl)
  by (subst (2) filter-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 · (smap · (Λ k · sdrop · k · xs) · (filter · (Λ x · prefix · (sdrop · x) · pat) · (supto · (MkI · (?patl-xsl + l)) · (MkI · xsl))))
    using False-
    (Θ 0 ≤ xsb) · (Θ 0 ≤ patb)
    by (subst filter-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)
next
  case FF — Recursive case: the lists generated by supto are too short
  have ?else = shead · (smap · (Λ x · eq-pat · (sdrop · x) · (filter · (Λ x · prefix · (sdrop · x) · pat) · (supto · (MkI · (?patl-xsl + l)) · (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) · (filter · (Λ x · prefix · (sdrop · x) · pat) · (supto · (MkI · (?patl-xsl + l)) · (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: shed-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-If 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 sfilt (!(Λ x. prefix-x-pat) oo stalls) to split, where “split-pat-x:s 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:] \rightarrow [:\'a:] \rightarrow [:\'a:] \times [:\'a:] where — Bird p128
[simp def]: split-pat-x:xs = If prefix-x:xs:pat then (xs, sdrop:(slength-x:xs)-pat) else split-pat-(stail-x:xs)

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

lemma split-bottom-iff[simp]: (split-pat-x:xs = ⊥) \iff (pat = ⊥ \or 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-x: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-x: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 ≠ ⊥
assumes $xs \neq \bot$
shows $\text{l}<\text{slencilh}-(\text{prod\_fst\ (split\_pat\cdot xs)})\cdot (\text{slencilh} \cdot xs + 1) = \text{TT}$
using \text{assms}
proof\(\text{(induct } xs\)\)
  case (\text{scons } x \ xs) then show ?case
    by (subst split\_unfold)
      (auto simp: \text{If2-def\_symmetric} one\_Integer\_def split: \text{If2-splits elim!}: \text{slencilh\ elim!}: \text{l<trans})
qed simp\_all
The predicate $p$ required by \text{Bird-strategy} is therefore \text{snull oo csnl}. It remains to find \text{op} and $z$ such that:

- \text{split\_pat\cdot [\cdot:] = z}
- \text{split\_pat\cdot (xs :@ [\cdot:]) = op\cdot (split\_pat\cdot xs)\cdot x}

so that $\text{split} = \text{sfoldl\_op\cdot z}$.

We obtain $z = ([\cdot:], \text{pat})$ directly from the definition of $\text{split}$. 

Bird derives $\text{op}$ on the basis of this crucial observation:

\textbf{lemma split\_snoc: — Bird p128}
\textbf{shows} $\text{split\_pat\cdot (xs :@ [\cdot:]) = split\_pat\cdot (\text{cfst\_pat\cdot (split\_pat\cdot xs)} :@ [\cdot:])}$
\textbf{proof(cases pat = \bot, simp, cases x = \bot, simp, induct xs)}
  case (\text{scons } x \ xs) then show ?case
    apply
      apply (subst (1 3) split\_unfold)
      apply (clarsimp simp: \text{If2-def\_symmetric} split: \text{If2-splits}; intro \text{conjI impI})
      apply (subst split\_unfold; fastforce)
      apply (subst split\_unfold; fastforce)
      apply (simp add: append\_prefixD)
    done
qed simp\_all

\textbf{fixrec — Bird p129}
\textbf{op :: [\cdot:a: Eq\_def::] \rightarrow [\cdot:a:] \times [\cdot:a:] \rightarrow [\cdot:a:] \times [\cdot:a:]}
\textbf{where}
\textbf{[simp def]}:
  \text{op\_pat\cdot (us, vs)\cdot x =}
    ( \text{If prefix-[\cdot:]\cdot vs then (us :@ [\cdot:], stail\_us)}
    else \text{If snull\_us then ([\cdot:], pat)}
    else \text{op\_pat\cdot (split\_pat\cdot (stail\_us))\cdot x})

\textbf{lemma op\_strict[simp]}:
  \text{op\_pat\cdot \bot = \bot}
  \text{op\_pat\cdot (us, \bot) = \bot}
  \text{op\_pat\cdot usus\cdot \bot = \bot}
\textbf{by} \text{fixrec\_simp+}

Bird demonstrates that \text{op} is partially correct wrt $\text{split}$, i.e., $\text{op\_pat\cdot split\_pat\cdot (xs :@ [\cdot:])} \subseteq \text{split\_pat\cdot (xs :@ [\cdot:])}$. For total correctness we essentially prove that $\text{op}$ terminates on well-defined arguments with an inductive argument.

\textbf{lemma op\_induct[case\_names step]}:
\textbf{fixes usus:: [\cdot:a:] \times 'b}
\textbf{assumes step:} $\forall usus. (\text{\text{l}(slencilh\ (\text{cfst\_usus}\cdot usus))} \cdot \text{slencilh\ (\text{cfst\_usus})}) = \text{TT} \implies \text{P usus')} \implies \text{P usus}
\textbf{shows} P usus
\textbf{proof(induct usus rule: wf\_induct)}
  let $\tau' = \{(\text{usus}', usus) | (\text{usus :: [\cdot:a:] \times 'b}) \cdot (\text{usus'} :: [\cdot:a:] \times 'b)\cdot \text{l}(\text{slencilh\ (\text{cfst\_usus})}) \cdot \text{slencilh\ (\text{cfst\_usus})}) = \text{TT}\}$
  show $\text{wf \tau'}$ 
  proof\(\text{(rule wf\_subset[OF wf\_inv\_image\_where f=lambda(x, -). slencilh\_x, OF wf\_subset[OF wf\_Integer\_ge\_less\_than[where d=0]])})$
let \(?rslen = \{(\text{slength-}us', \text{slength-}us) | (us :: [:\text{ac}]) (us' :: [:\text{ac}])\}. \text{lt}(\text{slength-}us')-(\text{slength-}us) = \text{TT}\) 
show \(?rslen \subseteq \text{Integer-ge-less-than} 0\) 
  apply (clarsimp simp: \text{Integer-ge-less-than-def} \text{zero-Integer-def}) 
  apply (metis \text{Integer.exhaust} \text{dist-eq-tr}(4) \text{dist-eq-tr}(6) \text{lt-Integer-bottom-iff} \text{lt-MkI-MkI} \text{slength-ge-0}) 
done
show \(?r \subseteq \text{inv-image} \?rslen (\lambda(x, -). \text{slength-}x)\) by (auto 0 3)

done

lemma \text{op-induct}[\text{case-names} \text{step}]:
  assumes \text{step}: \wedge \text{us}. (\wedge \text{us'}. \text{lt}(\text{slength-}us')-(\text{slength-}us) = \text{TT} \implies P \text{us'}) \implies P \text{us}
  shows P \text{us}
  by \ (\text{rule op-induct}[\text{where} P = P \circ \text{prod.fst} \text{ and} \text{usvs} = (\text{us}, \text{vs}) \text{ for} \text{vs::unit, simplified}])
  (\text{fastforce intro: \text{step}})

lemma \text{split-snoc-op}:
  \text{split-pat}(\text{xs} :: @ [:\text{x}]) = \text{op-pat}(\text{split-pat-}\text{xs})::\text{x}

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

lemma \text{split-sfoldl-op}:
  assumes \text{pat} \neq \bot
  shows \text{sfoldl}(\text{op-pat})::([], \text{pat}) = \text{split-pat} \ (\text{is} \ ?\text{lhs} = \ ?\text{rhs})
proof -
  have \ ?\text{lhs-}\text{xs} = \ ?\text{rhs-}\text{xs} \text{ for} \text{xs}
    using \text{assms} \text{ by (induct \text{xs rule: srev-induct}) \ (simp-all add: split-snoc-op)}
  then show \ ?\text{thesis} \text{ by (simp add: cfun-eq-iff)}
qed

lemma \text{matches-op}:
  shows \text{matches-pat} = \text{smap-cfst oo sfilter}(\text{snull oo csnd oo csnd})
    oo \text{sscanl}(\Lambda (n, \text{usvs}). x. (n + 1, \text{op-pat-usvs-x}))(0, ([::], \text{pat})) \ (\text{is} \ ?\text{lhs} = \ ?\text{rhs})
proof (\text{cases pat} = \bot)
  case True
  then have \ ?\text{lhs-}\text{xs} = \ ?\text{rhs-}\text{xs} \text{ for} \text{xs} \text{ by (cases \text{xs}; \text{clarsimp})}
  then show \ ?\text{thesis} \text{ by (simp add: cfun-eq-iff)}
qed
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 \texttt{split-sfoldl-op} we can rewrite \texttt{op} into a more perspicuous form that exhibits how KMP handles the failure of the text to continue matching the pattern:

\texttt{fixrec}

\[ op' :: [:'a:Eq-def:] \rightarrow [:'a:] \times [:'a:] \rightarrow 'a \rightarrow [:'a:] \times [:'a:] \]

\texttt{where}

\[ \texttt{simp def} : \]

\[\texttt{op''pat-(us, vs)-x =}
\begin{align*}
\text{If prefix-[:x:] in vs then (us @ [:x:], \texttt{stail-vs}) & --- continue matching} \\
\text{else If \texttt{snull}-us then ([:], \texttt{pat}) & --- fail at the start of the pattern: discard } x \\
\text{else sfoldl-(op''pat)-([]): pat)-(\texttt{stail-us} @ [:x:]) & --- fail later: discard \texttt{shead-us} and determine where to restart}
\end{align*}\]

Intuitively if \( x \) continues the pattern match then we extend the \texttt{split} of \texttt{pat} recorded in \texttt{us} and \texttt{vs}. Otherwise we need to find a prefix of \texttt{pat} to continue matching with. If we have yet to make any progress (i.e., \( \texttt{us} = [::] \)) we restart with the entire \texttt{pat} (aka \( z \)) and discard \( x \). Otherwise, because a match cannot begin with \( \texttt{us} @ [:x:] \), we \texttt{split pat} (aka \( z \)) by iterating \texttt{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 \texttt{op'} in HOL is tricky due to its use of higher-order nested recursion via \texttt{sfoldl}.

\texttt{lemma op''-strict[simp]}:

\[ \texttt{op''pat-\bot = \bot} \]

\[ \texttt{op''pat-(us, \bot) = \bot} \]

\[ \texttt{op''pat-usvs-\bot = \bot} \]

\texttt{by fixrec-simp+}

\texttt{lemma sfoldl-op''-strict[simp]}:

\[ \texttt{op''pat-(sfoldl-(op''pat)-(us, \bot)-xs)-x = \bot} \]

\texttt{by (induct xs arbitrary: x rule: srev-induct) simp-all}

\texttt{lemma op''-op:}

\texttt{shows op''pat-usvs-x = op-pat-usvs-x}

\texttt{proof(cases pat = \bot \ x = \bot \ 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 \( \ast : \texttt{sfoldl-(op''pat)-([]): pat)-xs = \texttt{sfoldl-(op-pat)-([]): pat)-xs} \)
    if \( \texttt{lt-(slength-xs)-(slength-(cfst-usvs)) = TT} \) for \( xs \)
    using that
  proof(induct xs rule: srev-induct)
  case (ssnoc(1,2,\_)) have \( \texttt{lt-(slength-xs')-(slength-(cfst-usvs)) = TT} \)
    using \( \texttt{lt-slength-0(2)} \) \( \texttt{lt-trans by auto} \)
    moreover
    from \( \texttt{step(2) ssnoc(1,2,\_)} \) have \( \texttt{lt-(slength-(cfst-(split-pat-xs')))-(slength-(cfst-usvs)) = TT} \)
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 \( sfoldl'-(op'-pat-)([:], \ pat-)(\mathit{stail-us} : @ [:x:]) \) computation in \( op' \) by finding a “representation” type \( t' \) for the “abstract” type \( [:'a:] \times [:'a:] \), a pair of functions \( \mathit{rep}, \ \mathit{abs} \) where \( \mathit{abs} \circ \mathit{rep} = \mathit{ID} \), and then finding a derived form of \( op' \) that works on \( t' \) rather than \( [:'a:] \times [:'a:] \). We also take the opportunity to add the ‘K’ optimisation in the form of the next function.

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

domain \( 'a \ \mathit{tree} \) — Bird p130

\[
= \mathit{Null}
\]

| Node (label :: \( 'a \)) (lazy left :: \( 'a \) tree) (lazy right :: \( 'a \) tree) — Strict in the label \( 'a \)

fixrec next :: \([[:'a:] \times [:'a:] : \mathit{tree}] \rightarrow ([[:'a:] \times [:'a:] : \mathit{tree}] \) tree where

\[
\begin{align*}
\text{next} \cdot :: & = t; \\
\end{align*}
\]

fixrec — Bird p131 “an even simpler form”, with the ‘K’ optimisation

\[
\begin{align*}
\mathit{root2} :: & = \mathit{rep2} \cdot \mathit{pat} ([:], \ \mathit{pat}) \\
\mathit{op2} :: & = \mathit{rep2} \cdot \mathit{tree} \rightarrow \mathit{tree} \rightarrow \mathit{root2} \cdot \mathit{tree} \\
\mathit{rep2} :: & = \mathit{next} \cdot \mathit{vs} \cdot \mathit{vs} \\
\mathit{right2} :: & = \mathit{right} \cdot \mathit{tree} \\
\end{align*}
\]

where

\[
\begin{align*}
\text{simp} & = \text{del}; \\
\text{root2-pat} & = \mathit{rep2-pat} ([:], \ \mathit{pat}) \\
\mathit{op2-pat} & = \mathit{root2-pat} \\
\mathit{usus} & = \mathit{next} \cdot \mathit{usus} \\
\text{simp} & = \text{del}; \\
\mathit{rep2-pat-usus} & = \mathit{usus} \cdot \mathit{usus} \cdot \mathit{next} \cdot \mathit{next} \cdot \mathit{next} \cdot \mathit{next} \\
\text{left2-pat} ([:], \ \mathit{vs}) & = \mathit{usus} \cdot \mathit{next} \cdot \mathit{usus} \\
\mathit{right2-pat} (us, [:]) & = \mathit{usus} \\
\text{simp} & = \text{del}; \\
\mathit{left2-pat} (us, us) & = \mathit{next} \cdot \mathit{next} \cdot \mathit{next} \cdot \mathit{next} \\
\text{right2-pat} (us, us) & = \mathit{usus} \\
\text{simp} & = \text{del}; \\
\mathit{usus} & = \mathit{next} \cdot \mathit{usus} \\
\end{align*}
\]

fixrec abs2 :: \( ([[:'a:] \times [:'a:] : \mathit{tree}] \rightarrow ([[:'a:] \times [:'a:] : \mathit{tree}] \) where

\[
\begin{align*}
\mathit{usus} & = \mathit{usus} \cdot \mathit{usus} \\
\end{align*}
\]
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 $q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_0 \xrightarrow{} \Box$. 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 (2012, §3.1) suggests it can be thought of as a doubly-linked list, following Takeichi and Akama (1991).
Bird’s suggestion (p134), our \textit{next} function is not recursive.

We note in passing that while MP only allows \textit{Null} on the left of the root node, \textit{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.

\textbf{schematic-goal} root2-op2-rep2-left2-right2-def: — Obtain the definition of these functions as a single fixed point

\begin{itemize}
  \item \texttt{root2} :: [':a:] \rightarrow ([':a:] \times [':a:]) tree
  \item \texttt{op2} :: [':a:] \rightarrow ([':a:] \times [':a:]) \rightarrow 'a \rightarrow ([':a:] \times [':a:]) tree
  \item \texttt{rep2} :: [':a:] \rightarrow [':a:] \rightarrow \rightarrow (\exists :a, \times [':a:] [':a:])
  \item \texttt{left2} :: [':a:] \rightarrow [':a:] \rightarrow \rightarrow (\exists :a, \times [':a:] [':a:])
  \item \texttt{right2} :: [':a:] \rightarrow [':a:] \rightarrow \rightarrow (\exists :a, \times [':a:] [':a:])
\end{itemize}

\[= \textit{fix}?F\]

\textbf{unfolding} \texttt{op2-def root2-def rep2-def left2-def right2-def} by \texttt{simp}

\textbf{lemma} \texttt{abs2-strict[simp]}:

\begin{itemize}
  \item \texttt{abs2} \cdot \perp = \perp
  \item \texttt{abs2} \cdot \texttt{Null} = \perp
\end{itemize}

by \texttt{fixrec-simp+}

\textbf{lemma} \texttt{next-strict[simp]}:

\begin{itemize}
  \item \texttt{next} \cdot \perp = \perp
  \item \texttt{next} \cdot \texttt{xs} \cdot \perp = \perp
  \item \texttt{next} \cdot (x :\# \texttt{xs}) \cdot \texttt{Node} \cdot (\texttt{us}, \perp) \cdot \texttt{l-r} = \perp
\end{itemize}

apply \texttt{fixrec-simp}

apply (cases \texttt{xs}, fixrec-simp; simp)

apply (cases \texttt{x} = \perp; cases \texttt{xs} = \perp; cases \texttt{us} = \perp; fixrec-simp)

done

\textbf{lemma} \texttt{next-Null[simp]}:

\begin{itemize}
  \item assumes \texttt{xs} \neq \perp
  \item shows \texttt{next} \cdot \texttt{xs} \cdot \texttt{Null} = \texttt{Null}
\end{itemize}

using \texttt{assms by (cases \texttt{xs}) simp-all}

\textbf{lemma} \texttt{next-snul[simp]}:

\begin{itemize}
  \item assumes \texttt{xs} \neq \perp
  \item shows \texttt{next} \cdot \texttt{xs} \cdot \texttt{Node} \cdot (\texttt{us}, [::]) \cdot \texttt{l-r} = \texttt{Node} \cdot (\texttt{us}, [::]) \cdot \texttt{l-r}
\end{itemize}

using \texttt{assms by (cases \texttt{xs}) simp-all}

\textbf{lemma} \texttt{op2-rep2-left2-right2-strict[simp]}:

\begin{itemize}
  \item \texttt{op2} \cdot \perp = \perp
  \item \texttt{op2} \cdot \texttt{Node} \cdot (\texttt{us}, \perp) \cdot \texttt{l-r} = \perp
  \item \texttt{op2} \cdot \texttt{Node} \cdot \texttt{usvs} \cdot \texttt{l-r} \cdot \perp = \perp
  \item \texttt{rep2} \cdot \perp = \perp
  \item \texttt{rep2} \cdot (\perp, \texttt{vs}) = \perp
  \item \texttt{rep2} \cdot (\texttt{us}, \perp) = \perp
  \item \texttt{right2} \cdot \texttt{pat} \cdot (\texttt{us}, \perp) = \perp
  \item \texttt{apply} fixrec-simp
  \item \texttt{apply} (cases \texttt{us} = \perp; fixrec-simp; simp)
  \item \texttt{apply} (cases \texttt{usvs} = \perp; fixrec-simp; simp)
  \item \texttt{apply} fixrec-simp
  \item \texttt{apply} fixrec-simp
  \item \texttt{apply} fixrec-simp
  \item \texttt{apply} fixrec-simp
\end{itemize}

done

\textbf{lemma} \texttt{snd-abs-root2-bottom[simp]}:

\texttt{prod} \cdot \texttt{snd} \cdot (\texttt{abs2} \cdot (\texttt{root2} \cdot \perp)) = \perp

by (simp add: root2_unfold rep2_unfold)

37
lemma abs-rep2-ID[simp]: $\text{abs2}(\text{rep2}\text{-}\text{pat}\_\text{usus}) = \text{usus}$
by (cases $\text{usus} = \bot$; subst $\text{rep2}\_\text{unfold}; \text{clarsimp}$)

lemma abs-rep2-ID: $\text{abs2 oo rep2}\_\text{pat} = \text{ID}$
by (clarsimp simp: cfun-eq-iff)

lemma rep2-snoc-right2: — Bird p131
assumes $\text{prefix}\_\text{us}\_\text{vs} : [x], \text{stail}\_\text{us} = \text{right2}\_\text{pat}\_\text{vs}$
shows $\text{rep2}\_\text{pat}(\text{next}\_\text{xs}(\text{rep2}\_\text{pat}\_\text{usus})\_\text{x}) = \text{op2}\_\text{pat}(\text{rep2}\_\text{pat}\_\text{usus})\_\text{x}$
proof —
  obtain $\text{us} \\text{vs}$ where $\text{usus} = (\text{us}, \text{vs})$ by force
  with assms show ?thesis
  by (cases $\text{xs}$; 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:

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

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

$$t = \text{Node}(\text{pat}, [::])\_\text{Null}\_\text{Null}$$

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 $\text{rep2}$. In pictures, we show that this diagram commutes:

Clearly this result self-composes: after an initial $\text{rep2}\_\text{pat}$ step, we can repeatedly simulate $\text{op}$ steps with $\text{op2}$ steps.

lemma op-op2-refinement:
assumes $\text{pat} \neq \bot$
shows $\text{rep2}\_\text{pat}(\text{op}\_\text{pat}\_\text{usus}\_\text{x}) = \text{op2}\_\text{pat}(\text{rep2}\_\text{pat}\_\text{usus})\_\text{x}$
proof(cases $x = \bot$ usus = $\bot$ rule: bool.exhaust[case-product bool.exhaust])
case False-False
then have $x \neq \bot$ usus $\neq \bot$ by simp-all
then show ?thesis
proof(induct usus arbitrary; x rule: op-induct)
case (step usus)
obtain $\text{us} \\text{vs}$ where $\text{usus} = (\text{us}, \text{vs})$ by fastforce
have $\ast$: $\text{sfoldl}(\text{op}\_\text{pat}(\text{root}\_\text{pat}\_\text{xs})\_\text{xs}) = \text{rep2}\_\text{pat}(\text{split}\_\text{pat}\_\text{xs})$ if $\text{lt}\_\text{(slength}\_\text{xs})(\text{slength}\_\text{us}) = \text{TT}$ for $x$
using that
proof(induct $\text{xs}$ rule: srev-induct)
case (ssnoc $\text{xs} \\text{xs}$)
  from ssnoc(1,2,4) have $\text{IH}$: $\text{sfoldl}(\text{op}\_\text{pat}(\text{root}\_\text{pat}\_\text{xs})\_\text{xs}) = \text{rep2}\_\text{pat}(\text{split}\_\text{pat}\_\text{xs})$ by — (rule ssnoc; auto intro: lt-trans dest: lt-slength-0)
obtain $\text{us} \\text{vs} \\text{where} \text{us/vs} = (\text{us}', \text{vs'})$ by fastforce
This representation works for automata with this sort of structure, but 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).

This computation can be thought of as a pair coroutines with a producer (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). 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

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

Thus: 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)

case False then show ?thesis

unfolding matches2.simps

apply (subt match-op) — Continue with previous derivation.

apply (subt 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
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.

cointuctive K :: ([a::Eq] × [a::]) tree ⇒ bool where
  K Null
| [ usus ≠ ⊥; K l; K r; ∨v vs. csnd-usus = v #: vs ⇒ l = Null ∨ (∃v' vs'. csnd-(label-l) = v' #: vs' ∧ eq-v-v' = FF) ] ⇒⇒ K (Node-usus-l-r)
decide K.intro[introl!, 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-op2-root2-rep2-split)
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, hide-lams) 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 us)
    case snil with scons splitl r show ?thesis
    by (clarsimp simp: split-sfoldl-op2-root2-rep2-split)
next
  case scons with ⟨pat ≠ ⊥⟩ ⟨us = u': # us'⟩ ⟨us' ≠ ⊥⟩ ⟨vs ≠ ⊥⟩ r splitl show ?thesis
  apply (clarsimp simp: split-sfoldl-op2-root2-rep2-split)
  apply (cases usl; cases usl; clarsimp simp: If2-def[symmetric] split-sfoldl-op2-root2-rep2-split split: If2-splits)
  apply (rename-tac ul' usl')
  apply (cut-tac usl'=prod.fst (split-pat-usl') and vsl=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

40
We retain

4.3 Step 3: Introduce an accumulating parameter (grep)

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

```plaintext
root3 :: [:a::Eq-def:] → ([[:a:] × [:a:]] tree
and op3 :: [:a:] → ([[:a:] × [:a:]] tree → :a → ([[:a:] × [:a:]] tree
and rep3 :: [:a:] → ([[:a:] × [:a:]] tree → [:a:] × [:a:] → ([[:a:] × [:a:]] tree
and grep3 :: [:a:] → ([[:a:] × [:a:]] tree → [:a:] × [:a:] → ([[:a:] × [:a:]] tree
where
```

[simp del]:

root3·pat = rep3·pat([[:, pat)
We further simplify by inlining

\[\text{where}\]

\[\text{and}\]

\[\text{and}\]

where

\[\text{schematic-goal} \quad \text{root3-op3-rep3-grep3-def:}\]

\[\text{schematic-goal} \quad \text{root4-op4-grep4-def:}\]

\[\text{schematic-goal} \quad \text{root4-op4-grep4-def:}\]

\[\text{schematic-goal} \quad \text{root3-op3-rep3-grep3-def:}\]

\[\text{schematic-goal} \quad \text{root3-op3-rep3-grep3-def:}\]

\[\text{lemma r3-2:}\]

\[\text{apply (simp add: match-nil-match-scons-list-case match-Null-match-Node-tree-case slist-case-distr tree-case-distr)}\]

\[\text{apply (simp add: fix-cprod fix-const)} \quad \text{— Very slow. Sensitive to tuple order due to the asymmetry of fix-cprod.}\]

\[\text{done}\]

4.4 Step 4: Inline rep

We further simplify by inlining rep3 into root3 and grep3.
\[ op_4 :: [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \rightarrow 'a \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \]
\[ grep_4 :: [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \rightarrow [\cdot a:] \times [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \)

\[ = \text{fix} \cdot \text{F} \]

**unfolding** root4-def op4-def grep4-def by simp

**lemma** fix-syn4-permute:

- **assumes** cont \((\lambda (X_1, X_2, X_3, X_4). F_1 X_1 X_2 X_3 X_4)\)
- **assumes** cont \((\lambda (X_1, X_2, X_3, X_4). F_2 X_1 X_2 X_3 X_4)\)
- **assumes** cont \((\lambda (X_1, X_2, X_3, X_4). F_3 X_1 X_2 X_3 X_4)\)
- **assumes** cont \((\lambda (X_1, X_2, X_3, X_4). F_4 X_1 X_2 X_3 X_4)\)
- **shows** fix-syn \((\lambda (X_1, X_2, X_3, X_4). (F_1 X_1 X_2 X_3 X_4, F_2 X_1 X_2 X_3 X_4, F_3 X_1 X_2 X_3 X_4, F_4 X_1 X_2 X_3 X_4))\)

\( = (\lambda (x_1, x_2, x_3, x_4). (x_1, x_2, x_3, x_4)) \)

**by** (induct rule: parallel-fix-ind) (use assms in (auto simp: prod-cont-iff))

**lemma** r4-3:

\( \begin{align*}
(root_4 & :: [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \\
op_4 & :: [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \rightarrow 'a \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \\
grep_4 & :: [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \rightarrow [\cdot a:] \times [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \\
= & \ (\Lambda (\text{root, op, rep, grep}). \text{root, op, grep}) \end{align*} \)

**unfolding** root3-op3-3-op3-3-op3-3-def root4-op4-op4-op4-def


**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 \(\text{cfst-usvs} :@ [\cdot v:]\) into \(\text{cfst-usvs}\).

**fixrec**

\[ root_5 :: [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \]

and \(\text{op}_5 :: [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \rightarrow 'a \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \)

and \(\text{grep}_5 :: [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \rightarrow [\cdot a:] \times [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \)

**where**

\[ \text{simp del]}: \]

- **root5-pat = grep5-pat-Null-([::], pat)\)
- **op5-pat-Null-x = root5-pat\)
- **usvs \neq \bot \implies \]**
- **op5-pat-(Node-usvs-l-r)-x = If prefix-[::]-(csnd-usvs) then r else op5-pat-l-x\)

\[ \text{simp del]}: \]

- **grep5-pat-l-usvs = Node-usvs-(next-(csnd-usvs)-l)-(case csnd-usvs of**
  \[ [[::]] \Rightarrow \text{Null} \]
  \[ v :# \ us \Rightarrow \text{grep5-pat-(case cfst-usvs of was cfst-usvs :@ [::v:]} \]
  \[ [[::]] \Rightarrow \text{root5-pat} \]
  \[ u :# \ us \Rightarrow \text{sfoldl-(op5-pat)-(root5-pat)-(us :@ [::v:])-(cfst-usvs :@ [::v:], us)} \]

**schematic-goal** root5-op5-grep5-def:

\( \begin{align*}
(root_5 & :: [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \\
op_5 & :: [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \rightarrow 'a \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \\
grep_5 & :: [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \rightarrow [\cdot a:] \times [\cdot a:] \rightarrow ([\cdot a:] \times [\cdot a:]) \quad \text{tree} \\
= & \text{fix} \cdot \text{F} \end{align*} \)

**unfolding** root5-def op5-def grep5-def by simp

**lemma** op5-grep5-strict[simp]:
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

root5 :: [\'a::Eq-def:] \to ([\'a:] \times ([\'a:]) tree

and op5 :: [\'a::Eq-def:] \to ([\'a:] \times ([\'a:]) tree \to 'a \to ([\'a:] \times ([\'a:]) tree

and grep5 :: [\'a:] \to ([\'a:] \times ([\'a:]) tree \to ([\'a:] \times ([\'a:]) tree

where

[simp del]:

root5-pat = grep5-pat-Null-([\[]], pat)

| op6-pat-Null-x = root6-pat

| usvs \not= \bot \implies

| op6-pat-(Node-usvs-l-r).x = If prefix-[\[]:(csnd-usvs) then r else op6-pat-l.x

| simp del:

grep6-pat-l-usus = Node-usus-(next-(csnd-usvs)-l)-(case csnd-usus of

\[\[] \Rightarrow Null

v :\# vs \Rightarrow grep6-pat-(op6-pat-l-v)-(cfst-usus @\[\[]; vs))

**schematic-goal** root6-op6-grep6-def:

( root6 :: [\'a::Eq-def:] \to ([\'a:] \times ([\'a:]) tree

, op6 :: [\'a:] \to ([\'a:] \times ([\'a:]) tree \to 'a \to ([\'a:] \times ([\'a:]) tree

, grep6 :: [\'a:] \to ([\'a:] \times ([\'a:]) tree \to ([\'a:] \times ([\'a:]) tree

= fixxF

**unfolding** root6-def op6-def grep6-def by simp

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

op6-pat-\bot = \bot

op6-pat-(Node-(us, \bot)-l-r) = \bot

op6-pat-(Node-usvs-l-r).\bot = \bot

grep6-pat-l-\bot = \bot

apply fixrec-simp

apply (cases us = \bot; fixrec-simp; simp)

apply (cases usvs = \bot; fixrec-simp; simp)

apply fixrec-simp

done

Intuitively this step cashes in the fact that, in the context of grep6-pat-l-usus, 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 \( \text{grep5-pat-l-usv} \) does not depend on \( l \) whereas that of \( \text{grep6-pat-l-usv} \) does, and therefore these cannot be extensionally equal. Furthermore the computations of the corresponding roots do not proceed in lockstep: consider the computation of the left subtree.

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

- \( R \perp \perp \)
- \( R \text{ Null Null} \) and
- if \( R (\text{Node} \cdot x \cdot l \cdot r) (\text{Node} \cdot x' \cdot l' \cdot r') \) then \( x = x', R l l', \) and \( R r r' \).

The following pair of \( \text{left} \) functions define suitable left paths from the corresponding \( \text{roots} \).

\[
\text{fixrec left5} :: [+a::Eq-def:] \rightarrow [+a:] \rightarrow ([+a:] \times [+a:]) \text{ tree where}
\]
\[
\text{left5-pat}::[+a] = \text{Null}
| [+u \neq \perp; us \neq \perp] \implies
\text{left5-pat}(u :: \# us) = \text{sfoldl}(\text{op5-pat})\cdot(\text{root5-pat})\cdot us
\]

\[
\text{fixrec left6} :: [+a::Eq-def:] \rightarrow [+a:] \rightarrow ([+a:] \times [+a:]) \text{ tree where}
\]
\[
\text{left6-pat}::[+a] = \text{Null}
| [+u \neq \perp; us \neq \perp] \implies
\text{left6-pat}(u :: \# us) = \text{sfoldl}(\text{op6-pat})\cdot(\text{root6-pat})\cdot us
\]

\[
\text{inductive — This relation is not inductive.}
\]
\[
\text{root-bisim} :: [+a::Eq-def:] \Rightarrow ([+a:] \times [+a:]) \text{ tree} \Rightarrow ([+a:] \times [+a:]) \text{ tree} \Rightarrow \text{bool}
\]
\[
\text{for pat :: [+a]}
\]

\[
\text{where}
\]
\[
\text{bottom}: \text{root-bisim pat} \perp \perp
| \text{Null}: \text{root-bisim pat Null Null}
| \text{gl}: [\text{pat} \neq \perp; us \neq \perp; vs \neq \perp]
\implies \text{root-bisim pat} (\text{grep6-pat}\cdot(\text{left6-pat-us})\cdot(\text{us}, vs)) (\text{grep5-pat}\cdot(\text{left5-pat-us})\cdot(\text{us}, vs))
\]

\[
\text{declare root-bisim.intro, intro!, simp}
\]

\[
\text{lemma left6-left5-strict[simp]}:
\text{left6-pat}\perp = \perp
\text{left5-pat}\perp = \perp
\]
by \text{fixrec-simp+}

\[
\text{lemma op6-left6: [us \neq \perp; v \neq \perp] \implies op6-pat\cdot(\text{left6-pat-us})\cdot v = \text{left6-pat\cdot(\text{us}:@ [+v:])}}
\]
by \text{(cases us) simp-all)

\[
\text{lemma op5-left5: [us \neq \perp; v \neq \perp] \implies op5-pat\cdot(\text{left5-pat-us})\cdot v = \text{left5-pat\cdot(\text{us}:@ [+v:])}}
\]
by \text{(cases us) simp-all)

\[
\text{lemma root5-left5: v \neq \perp \implies root5-pat = left5-pat::[v:]}\]
by \text{simp}

\[
\text{lemma op5-sfoldl-left5: [us \neq \perp; u \neq \perp; v \neq \perp] \implies}
\text{op5-pat\cdot(\text{sfoldl}\cdot(\text{op5-pat})\cdot(\text{root5-pat-us})\cdot v = \text{left5-pat\cdot(\text{u}:@ us:@ [+v:]))}}
\]
by \text{simp}

\[
\text{lemma root-bisim-root:}
\]
\[
\text{assumes pat \neq \perp}
\]
The main part of establishing that \textit{root-bisim} is a tree-\textit{bisim} is in showing that the left paths constructed by the \textit{greps} are \textit{root-bisim}-related. We do this by inducting over the length of the pattern so far matched \textit{(us)}, as we did when proving that this tree has the ‘K’ property in §4.2.

\begin{verbatim}
lemma next-grep6-cases[consumes 3, case-names gl nl]:
    assumes vs ≠ ⊥
    assumes xs ≠ ⊥
    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-[:x]:xs = FF
  shows op6-pat (next-xs t6):x = op6-pat t6:x ∧ op5-pat (next-xs t5):x = op5-pat t5: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 = ⊥, 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
\end{verbatim}
\{ case (1 usvs l r vs x)
   from rbl
   have L: \text{le}-(\text{slength}\cdot(\text{prod} \cdot \text{fst usvs}'))-(\text{slength} \cdot \text{us} + 1) = \text{TT}
     if op6-pat-(next-us\cdot(\text{left6} \cdot \text{us}))-x = \text{Node-usvs'} l r
     and cfst-usvs' \neq \bot
     and vs \neq \bot
     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 \neq \bot) have \text{le}-(\text{slength}\cdot(\text{cfst-usvs}'))-(\text{slength} \cdot \text{us}'' + 1) = \text{TT}
         apply simp
         apply (subst (asm) (2) grep6.unfold)
         apply (fastforce dest: step(2, 3)[rotated])
         done
       moreover from gl scons have \text{lt}-(\text{slength} \cdot \text{us}''')-(\text{slength} \cdot \text{stail-us} + 1) = \text{TT}
         apply simp
         apply (subst (asm) grep6.unfold)
         apply (fastforce dest: step(3)[rotated])
         done
       ultimately show ?thesis
       apply clarsimp
       apply (metis \text{Integer-le-both-plus-1} \text{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-\text{Integer}
     sappend-snil-id-right \text{slength}.simps(2) \text{slength-bottom-iff slength-sappend slist.con-rews(3) tree-injects'})
   apply (rule L; fastforce)
   done \}
note \text{slength-ogl = this}
\{ case (2 usvs l r vs x xs)
   from 2 have xs \neq \bot by clarsimp
   from (vs \neq \bot) (xs \neq \bot) 2(1) show ?case
   proof(cases rule: next-grep6-cases)

case $gl$ with $\langle \text{cfst} \cdot \text{usvs} \neq \bot \rangle$ $\langle x \neq \bot \rangle$ show ?thesis using slength-ogl by blast

next

case $nl$

from rbl show ?thesis

proof cases

case bottom with $nl$ $\langle \text{cfst} \cdot \text{usvs} \neq \bot \rangle$ show ?thesis by simp

next

case Null with $nl$ $\langle us \neq \bot \rangle$ $\langle vs \neq \bot \rangle$ show ?thesis

proof cases

case bottom with $nl$ $\langle \text{cfst} \cdot \text{usvs} \neq \bot \rangle$ show ?thesis by simp

next

case $(gl \text{ us}'' \text{ vs}'')$ show ?thesis

proof (cases $us$)

case bottom with $\langle us \neq \bot \rangle$ $\langle vs \neq \bot \rangle$ show ?thesis by simp

next

case $\text{snl}$ with $gl$ show ?thesis by $(\text{subst (asm) grep6.unfold})$ simp

next

case $(\text{scons u' us'})$ with $2$ $nl$ $gl$ show ?thesis

apply clarsimp

apply $(\text{subst (asm) grep6.unfold})$

apply clarsimp

apply $(\text{drule step(3)[rotated]; clarsimp})$

apply $(\text{drule step(2)[rotated]; clarsimp})$

apply $(\text{clarsimp simp: zero-Integer-def one-Integer-def elim!: slengthE})$

done

next

case $(\text{ssnoc u' us'})$

then have root-bisim pat $\langle \text{for6 us' } \langle \text{for5 us'} \rangle \rangle$ by $(\text{fastforce intro: step(5)})$

then show ?thesis

proof cases

case bottom with $3$ $\text{ssnoc}$ show ?thesis by simp

next

case Null with $3$ $\text{ssnoc}$ show ?thesis

apply simp

apply $(\text{subst (asm) grep6.unfold})$

apply $(\text{subst (asm) grep6.unfold})$

apply $(\text{clarsimp simp: zero-Integer-def one-Integer-def elim!: slengthE})$

done

next

case $(gl \text{ us}'' \text{ vs}'')$ with $3$ $\text{ssnoc}$ show ?thesis

apply clarsimp

apply $(\text{subst (asm) grep6.unfold})$

apply $(\text{fastcall simp: zero-Integer-def one-Integer-def split: if-splits dest!: step(1)[rotated] step(3)[rotated] elim!: slengthE})$
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)
    using rbl
    apply (auto simp: root-bisim-op-next56 elim!: root-bisim.cases intro: root-bisim-root)
    apply (subst (asm) grep6.unfold)
    apply (cases vs; clarsimp simp del: left6.simps left5.simps simp add: root5-left5)
    apply (metis (no-types, lifting) op5-sfoldl-left root5-left sappend-bottom-iff slist.con-rews(3) slist.con-rews(4))
  done
qed

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 vs;clarsimp simp del: left6.simps left5.simps simp add: root5-left5)
    apply (metis (no-types, lifting) op5-sfoldl-left root5-left sappend-bottom-iff slist.con-rews(3) slist.con-rews(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 (clarsimp intro: step(5))
  then show ?thesis
    proof cases
      case (gl us'' vs'') with ssnoc show ?thesis
        apply clarsimp
        apply (subst (asm) grep6.unfold)
        apply (clarsimp 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 rbl: ?rbl 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 simp: cases us'';clarsimp simp)
    apply (metis Rep-cfun-strict1 bottom left5.simps(2) left6.simps(2) next-snil next-strict(1) rbl)
    apply (cases vs;clarsimp simp)

49
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 \(\text{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

\[
\begin{align*}
\text{root7} &:: :('[a]:\text{Eq-def}:) \rightarrow ([':[a] \times :'[a]) \text{tree} \\
\text{and op7} &:: :('[a]:\text{Eq-def}:) \rightarrow ([':[a] \times :'[a]) \text{tree} \rightarrow ':a \rightarrow ([':[a] \times :'[a]) \text{tree} \\
\text{and grep7} &:: :'[a] \rightarrow ([':[a] \times :'[a]) \text{tree} \rightarrow :'[a] \times :'[a] \rightarrow ([':[a] \times :'[a]) \text{tree}
\end{align*}
\]

where

\[
\begin{align*}
\text{simp del:} \\
\text{root7-pat} = \text{grep7-pat-Null-}([[:]], \text{pat}) \\
\text{| op7-pat-Null-}x = \text{root7-pat} \\
\text{| op7-pat-(Node-}x:(us, [[:]])\cdot(\text{r-l})\cdot x = \text{op7-pat-}l\cdot x \quad \text{Unfold prefix} \\
\text{| [v \neq \bot; \text{vs} \neq \bot] \implies } \text{op7-pat-(Node-}x:(us, v :\# \text{vs})\cdot(\text{r-l})\cdot x = \text{If eq-x-v then r else op7-pat-}l\cdot x \\
\text{| simp del:} \\
\text{grep7-pat-}l\cdot(\text{us, [[:]])} = \text{Node-}(\text{us, [[:]])\cdot(\text{r-l})-\text{Null} \quad \text{Case split on vs hoisted above Node.} \\
\text{| [v \neq \bot; \text{vs} \neq \bot] \implies } \text{grep7-pat-}l\cdot(\text{us, v :\# vs}) = \text{Node-}(\text{us, v :\# vs})\cdot(\text{next-(v :\# vs)-l})\cdot(\text{grep7-pat-}(\text{op7-pat-}l\cdot v))\cdot(\text{us :\@ [[:]], vs})
\end{align*}
\]

schematic-goal root7-op7-grep7-def:

\[
\begin{align*}
\text{root7} &:: :('[a]:\text{Eq-def}:) \rightarrow ([':[a] \times :'[a]) \text{tree} \\
\text{op7} &:: :'[a] \rightarrow ([':[a] \times :'[a]) \text{tree} \rightarrow ':a \rightarrow ([':[a] \times :'[a]) \text{tree} \\
\text{grep7} &:: :'[a] \rightarrow ([':[a] \times :'[a]) \text{tree} \rightarrow :'[a] \times :'[a] \rightarrow ([':[a] \times :'[a]) \text{tree} \\
= \text{fix-x-}F
\end{align*}
\]

unfolding root7-def op7-def grep7-def by simp

lemma r7-6-aux:

assumes pat \neq \bot

shows

\[
\begin{align*}
(\Lambda (\text{root, op, grep}). (\text{root-pat, seq-x-}(\text{op-pat-t-x}), \text{grep-pat-l-(us, vs)))) \\
(\text{root7} &:: :('[a]:\text{Eq-def}:) \rightarrow ([':[a] \times :'[a]) \text{tree} \\
\text{op7} &:: :'[a] \rightarrow ([':[a] \times :'[a]) \text{tree} \rightarrow ':a \rightarrow ([':[a] \times :'[a]) \text{tree} \\
\text{grep7} &:: :'[a] \rightarrow ([':[a] \times :'[a]) \text{tree} \rightarrow :'[a] \times :'[a] \rightarrow ([':[a] \times :'[a]) \text{tree} \\
= (\Lambda (\text{root, op, grep}). (\text{root-pat, seq-x-}(\text{op-pat-t-x}), \text{seq-vs-}(\text{grep-pat-l-}(\text{us, vs}))))
\end{align*}
\]

unfolding root6-op6-grep6-def root7-op7-grep7-def

proof(induct arbitrary: t x l us \text{vs rule: parallel-fix-ind(case-names adm bottom step))}

case (step X Y t \text{x l us \text{vs then show \text{?case}}}

apply \_ \\
apply \text{(cases X, cases Y)}

apply (rename-tac r10 a10 g10 r9 o9 g9)


apply (intro all conj)

apply (case-tac t \text{; clarisimp})

apply (rename-tac \text{us \text{vs l r}})

apply (case-tac x = \_\; \text{; clarisimp})

apply (case-tac vs \text{; clarisimp; fail})

apply (case-tac vs; clarisimp)

apply (metis ID1 seq-simps(3))

done

qed simp-all
We now discard us.

This is essentially another data refinement.

fixrec next': 'a::Eq-def → ['a:] tree → ['a:] tree where
next'-x-Null = Null
next'-x-(Node-['l:]::'l-r) = Node-['l:]::l-r

next'-x-(Node-(v # vs)::l-r) = If eq-x-v then l else Node-(v # vs)::l-r

fixrec — Bird p133
root8 :: ['a::Eq-def:] → ['a:] tree
and op8 :: ['a:] → ['a:] tree → 'a → ['a:] tree
and grep8 :: ['a:] → ['a:] tree → ['a:] → ['a:] tree

where
[simp del]:
root8-pat = grep8-pat-Null-pat

| op8-pat-Null-x = root8-pat
| op8-pat-(Node-['l:]::'l-r)::x = op8-pat-::l-x

| [v ≠ ⊥; vs ≠ ⊥] ⇒
op8-pat-(Node-(v # vs)::l-r)::x = If eq-x-v then r else op8-pat-::l-x

| grep8-pat-l-['l:] = Node-['l:]::l-Null

| [v ≠ ⊥; vs ≠ ⊥] ⇒
grep8-pat-l(v # vs) = Node-(v # vs)(next'-v.l)(grep8-pat-(op8-pat-::l-v)::vs)

fixrec ok8 :: ['a:] tree → tr where

vs ≠ ⊥ ⇒ ok8-(Node-::vs-::l-r) = snull::vs

schematic-goal root8-op8-grep8-def:
( root8 :: ['a::Eq-def:] → ['a:] tree
  , op8 :: ['a:] → ['a:] tree → 'a → ['a:] tree
  , grep8 :: ['a:] → ['a:] tree → ['a:] → ['a:] tree )
= fix-?F

unfolding op8-def root8-def grep8-def by simp

lemma next'-strict[simp]:

next'-x-⊥ = ⊥
next'-⊥-(Node-(v # vs)::l-r) = ⊥

by (cases v :: vs = ⊥; 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 $v \neq \bot$

assumes $vs \neq \bot$

shows $next'(v \cdot (\text{tree-map}' \cdot \text{csnd} \cdot t)) = \text{tree-map}' \cdot \text{csnd} \cdot (next\cdot(v \#: vs) \cdot t)$

**Proof** (induct $t$)

- case $(\text{Node } usvs' \ l \ r)$ with $\text{assms show } ?\text{case}$
  - apply $(\text{cases usvs'; clarsimp})$
  - apply $(\text{rename-tac us'' vs''})$
  - apply $(\text{case-tac vs''; clarsimp simp: If-distr})$
  - done

**Qed** (use $\text{assms in simp-all}$)

**Lemma** $r8-7[^{\text{simplified}}]$: 

shows $(\Lambda (\text{root}, \text{op}, \text{grep}). (\text{root-pat})$

, $\text{op-pat} \cdot (\text{tree-map}' \cdot \text{csnd} \cdot t) \cdot x$

, $\text{grep-pat} \cdot (\text{tree-map}' \cdot \text{csnd} \cdot l) \cdot (\text{csnd} \cdot usvs)) \cdot (\text{root8}, \text{op8}, \text{grep8})$

$= (\Lambda (\text{root}, \text{op}, \text{grep}). (\text{tree-map}' \cdot \text{csnd} \cdot (\text{root-pat})$

, $\text{tree-map}' \cdot \text{csnd} \cdot (\text{op-pat} \cdot t \cdot x)$

, $\text{tree-map}' \cdot \text{csnd} \cdot (\text{grep-pat} \cdot \text{csnd} \cdot usvs)) \cdot (\text{root7}, \text{op7}, \text{grep7})$

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 $(\text{step } X \ Y \ l \ t \ usvs \ x)$ then show $?\text{case}$
  - apply $-$
  - apply $(\text{cases } X; \text{cases } Y)$
  - apply $(\text{clarsimp simp: cfuns-eq-iff next'-next}$
    - match-snilt-match-scons-list-case slist-case-distr
    - cong: slist-case-cong)
  - apply $(\text{cases } t; \text{clarsimp simp: If-distr})$
  - apply $(\text{rename-tac us vs l r})$
  - apply $(\text{case-tac vs; fastforce})$
  - 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 $x \neq \bot$:

\[
\begin{array}{ccc}
a & \xrightarrow{\Lambda a. f \cdot a \cdot x} & a' \\
\downarrow g & & \downarrow g \\
b & \xrightarrow{\Lambda a. f' \cdot a \cdot x} & b'
\end{array}
\]

**Lemma** ok8-ok8: ok8 oo tree-map' csnd = snull oo csnd oo abs2 (is $?\text{lhs} = ?\text{rhs}$)

**Proof** (rule cfuns-eqI)

- fix $t$ show $?\text{lhs} \cdot t = ?\text{rhs} \cdot t$
  - by $(\text{cases } t; \text{clarsimp})$ (metis ok8.simps ok8-strict(I) snull-strict tree.con-rews(3))

**Qed**

**Lemma** matches8: — Bird p133

- shows matches-pat = smap-cfst oo sfilter • (ok8 oo csnd) oo sscanl • (\lambda (n, x) y. (n + 1, op8-pat \cdot x \cdot y)) • (0, root8-pat)
  - is $?\text{lhs} = ?\text{rhs}$

**Proof** (cases pat $= \bot$)

- case True

53
then have \( \text{ths}-xs = \text{rhs}-xs \) for \( xs \) by (cases \( xs; \text{clarsimp} \))
then show \( \text{thesis} \) by (simp add: cfun-eq-iff)
next
  case False
  then have \( *: \text{matches-pat} = \text{smap-cfst oo sfilter}-(\text{snull} oo \text{csnd} oo \text{abs2} oo \text{csnd}) oo \text{sscanl}-(\Lambda (n, x) y. (n+1, \text{op7-pat}-x-y))(0, \text{root7-pat}) \)
    using data-refinement\( [\text{where} \ 'a'=\text{'}a]\) r3-2\( [\text{where} \ 'a'=\text{'}a]\) r4-3\( [\text{where} \ 'a'=\text{'}a]\) r5-4\( [\text{where} \ 'a'=\text{'}a]\) r6-5(1)\( [\text{where} \ \text{pat}=\text{pat}] \) r7-6\( [\text{where} \ \text{pat}=\text{pat}] \)
    unfolding matches2.unfold by (fastforce simp: oo-assoc cfun-eq-iff csplit-def intro: cfun-arg-cong sscanl-cong)
  from (\( \text{pat} \neq \bot \)) show \( \text{thesis} \)
    apply \－
    apply (subst conjunct1[OF r8-7])
    apply (subst sscanl-lfp-fusion\( [\text{where} \ g=\text{ID} ** \text{tree-map'}\text{csnd and} \ z=(0, \text{root7-pat}), \text{simplified}, \text{symmetric}] \))
    prefer 2
    apply (subst oo-assoc, subst sfilter-smap, simp)
    apply (simp add: oo-associative)
    apply (simp add: oo-associative[\text{symmetric}])
    apply (subst oo-assoc, subst ok8-ok8)
    apply (clarsimp simp: oo-associative *)
    apply (rule refl)
    apply (clarsimp simp: r8-7)
  done
qed

4.9 Step 9: Factor out \text{pat} (final version)

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

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

\textbf{fixrec} \text{matchesf} :: \( [::\text{'}a::\text{Eq-def}:] \rightarrow [::\text{'}a:] \rightarrow [:\text{Integer}:] \) \text{where}

\text{simp def}: \text{matchesf-pat} =
  (\text{Letrec} \text{okf} = (\Lambda (\text{Node-vs-l-r}). \text{snull-vs});
   \text{nextf} = (\Lambda t x. \text{case} t \text{ of}
    \text{Null} \Rightarrow \text{Null}
    | \text{Node-vs-l-r} \Rightarrow (\text{case} vs \text{ of}
      [::] \Rightarrow t
    | v :# vs \Rightarrow \text{If eq}-x\cdot v \text{ then} l \text{ else} t));
   \text{rootf} = \text{grepf-Null-pat},
   \text{opf} = (\Lambda t x. \text{case} t \text{ of}
    \text{Null} \Rightarrow \text{rootf}
    | \text{Node-vs-l-r} \Rightarrow (\text{case} vs \text{ of}
      [::] \Rightarrow \text{opf-l-x}
    | v :# vs \Rightarrow \text{If eq}-x\cdot v \text{ then} r \text{ else} \text{opf-l-x})),
   \text{grepf} = (\Lambda l vs. \text{case} vs \text{ of}
    [::] \Rightarrow \text{Node-}[::]\cdot l\cdot \text{Null}
    | v :# vs \Rightarrow \text{Node-(v :# vs)-(nextf-v-l)-(grepf-(opf-l-v)-vs)});
   \text{stepf} = (\Lambda (n, t) x. (n+1, \text{opf-t-x}))
  )
in \text{smap-cfst oo sfilter-(okf oo csnd) oo sscanl-stepf-(0, \text{rootf})}

\textbf{lemma} \text{matchesf-ok8}: (\Lambda (\text{Node-vs-l-r}). \text{snull-vs}) = \text{ok8}
by (\text{clarsimp simp: cfun-eq-iff}; \text{rename-tac} x; \text{case-tac} x; \text{clarsimp})

\textbf{lemma} \text{matchesf-next'}:
(\Lambda t x. \text{case} t \text{ of} \text{Null} \Rightarrow \text{Null} | \text{Node-vs-l-r} \Rightarrow (\text{case} vs \text{ of} [::] \Rightarrow t | v :# vs \Rightarrow \text{If eq}-x\cdot v \text{ then} l \text{ else} t)) = \text{next}'
apply (\text{clarsimp simp: cfun-eq-iff next'}\cdot \text{unfold}
    \text{match-snil-match-scons-slist-case slist-case-distr}
apply (simp cong: tree-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

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 xx. case xx 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 xx from assms show ?lhs·xx = ?rhs·xx
  by (induct xx 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


% -- mode: prolog --
% Bird’s Morris-Pratt string matcher
% - 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(_, 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(_, N, T, [], Ns) :- ( ok(T) -> Ns = [N] ; Ns = []).
matches_aux(Root, N, T, [X|Xs], Ns) :-
  Ni is N + 1, op(Root, T, X, T1),
  ( ok(T) -> (Ns = [N|Ns1], matches_aux(Root, Ni, T1, Xs, Ns1))
   ; matches_aux(Root, Ni, 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], [1,2,1,2,3,1,2,3,1,2,3,1,2], Ns).

Figure 3: The final KMP program transliterated into Prolog.
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 =
      (case t of
        Null => (0, t)
      | Node ([], _, r) => if ok t then (n + rheight r)
        else driver ((n, t) + 1, root)
      | Node (v :: vs, l, r) =>
        let val nt' = step (n, t) x
        in if ok t then n :: driver nt' xs else driver nt' xs end
      end)
    val () = root := Thunk (fn () => raise Fail "blackhole")
    fun rheight (t: 'a tree) =
      case t of Null => 0
      | Node (_, l, 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
      end in
        driver (0, root)
      end
  end
end;

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

58