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

Peter Gammie

March 17, 2025

Abstract

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

Contents

1 Introduction

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

```
module KMP where

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

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

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

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

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

    root = grep Null ws

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

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

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

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

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

    root = grep Null ws

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

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

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

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

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

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

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

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

We discuss related work in §5.

1.1 Formal setting

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

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

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

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

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

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

2 Extra HOLCF

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

lemma *predE*:
  obtains (*strict*) $p{\cdot}\bot = \bot$ | (*FF*) $p = (\Lambda\ x.\ FF)$ | (*TT*) $p = (\Lambda\ x.\ TT)$
⟨*proof*⟩

lemma *retraction-cfcomp-strict*:
  assumes $f\ oo\ g = ID$
  shows $f{\cdot}\bot = \bot$
⟨*proof*⟩

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

⟨*proof*⟩

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

lemma *If-cancel*[*simp*]: (*If b then x else x*) = *seq*·*b*·*x*
⟨*proof*⟩

lemma *seq-below*[*iff*]: *seq*·*x*·*y* ⊑ *y*
⟨*proof*⟩

lemma *seq-strict-distr*: *f*·⊥ = ⊥ ⟹ *seq*·*x*·(*f*·*y*) = *f*·(*seq*·*x*·*y*)
⟨*proof*⟩

lemma *strictify-below*[*iff*]: *strictify*·*f* ⊑ *f*
⟨*proof*⟩

lemma *If-distr*:
⟦*f* ⊥ = ⊥; *cont f*⟧ ⟹ *f* (*If b then t else e*) = (*If b then f t else f e*)
⟦*cont t*′; *cont e*′⟧ ⟹ (*If b then t*′ *else e*′) *x* = (*If b then t*′ *x else e*′ *x*)
(*If b then t*‴ *else e*‴)·*x* = (*If b then t*‴·*x else e*‴·*x*)
⟦*g* ⊥ = ⊥; *cont g*⟧ ⟹ *g* (*If b then t*″ *else e*″) *y* = (*If b then g t*″ *y else g e*″ *y*)
⟨*proof*⟩

lemma *If2-split-asm*: *P* (*If2 Q x y*) ⟷ ¬(*Q* = ⊥ ∧ ¬*P* ⊥ ∨ *Q* = *TT* ∧ ¬*P x* ∨ *Q* = *FF* ∧ ¬*P y*)
⟨*proof*⟩

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

lemma *If2-cont*[*simp*, *cont2cont*]:
assumes *cont i*
assumes *cont t*
assumes *cont e*
shows *cont* (λ*x*. *If2* (*i x*) (*t x*) (*e x*))
⟨*proof*⟩

lemma *If-else-FF*[*simp*]: (*If b then t else FF*) = (*b andalso t*)
⟨*proof*⟩

lemma *If-then-TT*[*simp*]: (*If b then TT else e*) = (*b orelse e*)
⟨*proof*⟩

lemma *If-cong*:
assumes *b* = *b*′
assumes*b* = *TT* ⟹ *t* = *t*′
assumes *b* = *FF* ⟹ *e* = *e*′
shows (*If b then t else e*) = (*If b*′ *then t*′ *else e*′)
⟨*proof*⟩

lemma *If-tr*: (*If b then t else e*) = ((*b andalso t*) *orelse* (*neg*·*b andalso e*))
⟨*proof*⟩

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

lemma *If-else-absorb*:
assumes *c* = ⊥ ⟹ *e* = ⊥
assumes *c* = *TT* ⟹ *e* = *t*

shows *If c then t else e = e*
⟨*proof*⟩

lemma *andalso-cong*: ⟦*P = P'*; *P' = TT* ⟹ *Q = Q'*⟧ ⟹ (*P andalso Q*) = (*P' andalso Q'*)
⟨*proof*⟩

lemma *andalso-weaken-left*:
assumes *P = TT* ⟹ *Q = TT*
assumes *P = FF* ⟹ *Q* ≠ ⊥
assumes *P* = ⊥ ⟹ *Q* ≠ *FF*
shows *P* = (*Q andalso P*)
⟨*proof*⟩

lemma *orelse-cong*: ⟦*P = P'*; *P' = FF* ⟹ *Q = Q'*⟧ ⟹ (*P orelse Q*) = (*P' orelse Q'*)
⟨*proof*⟩

lemma *orelse-conv*[*simp*]:
((*x orelse y*) = *TT*) ⟷ (*x = TT* ∨ (*x = FF* ∧ *y = TT*))
((*x orelse y*) = ⊥) ⟷ (*x* = ⊥ ∨ (*x = FF* ∧ *y* = ⊥))
⟨*proof*⟩

lemma *csplit-cfun2*: *cont F* ⟹ (Λ *x*. *F x*) = (Λ (*x*, *y*). *F* (*x*, *y*))
⟨*proof*⟩

lemma *csplit-cfun3*: *cont F* ⟹ (Λ *x*. *F x*) = (Λ (*x*, *y*, *z*). *F* (*x*, *y*, *z*))
⟨*proof*⟩

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·⊥·⊥ = ⊥
⟨*proof*⟩

lemma *convol-simp*[*simp*]: (*f* && *g*)·*x* = (*f*·*x*, *g*·*x*)
⟨*proof*⟩

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*)
⟨*proof*⟩

lemma *map-prod-ID*[*simp*]: *ID* ** *ID* = *ID*
⟨*proof*⟩

lemma *map-prod-app*[*simp*]: (*f* ** *g*)·*x* = (*f*·(*cfst*·*x*), *g*·(*csnd*·*x*))
⟨*proof*⟩

lemma *map-prod-cfst*[*simp*]: *cfst oo* (*f* ** *g*) = *f oo cfst*
⟨*proof*⟩

lemma *map-prod-csnd*[*simp*]: *csnd oo* (*f* ⋆⋆ *g*) = *g oo csnd*
⟨*proof*⟩

2.1 Extra HOLCF Prelude.

lemma *eq-strict*[*simp*]: *eq*·(⊥::′*a*::*Eq-strict*) = ⊥
⟨*proof*⟩

lemma *Integer-le-both-plus-1*[*simp*]:
fixes *m* :: *Integer*
shows *le*·(*m* + *1*)·(*n* + *1*) = *le*·*m*·*n*
⟨*proof*⟩

lemma *plus-eq-MkI-conv*:
l + *n* = *MkI*·*m* ⟷ (∃ *l*′ *n*′. *l* = *MkI*·*l*′ ∧ *n* = *MkI*·*n*′ ∧ *m* = *l*′ + *n*′)
⟨*proof*⟩

lemma *lt-defined*:
fixes *x* :: *Integer*
shows
lt·*x*·*y* = *TT* ⟹ (*x* ≠ ⊥ ∧ *y* ≠ ⊥)
lt·*x*·*y* = *FF* ⟹ (*x* ≠ ⊥ ∧ *y* ≠ ⊥)
⟨*proof*⟩

lemma *le-defined*:
fixes *x* :: *Integer*
shows
le·*x*·*y* = *TT* ⟹ (*x* ≠ ⊥ ∧ *y* ≠ ⊥)
le·*x*·*y* = *FF* ⟹ (*x* ≠ ⊥ ∧ *y* ≠ ⊥)
⟨*proof*⟩

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*⟩

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* = ⊥)
⟨*proof*⟩

lemma *eq-defined-reflD*[*simp*]:
(*eq*·*a*·*a* = *TT*) ⟷ *a* ≠ ⊥
(*TT* = *eq*·*a*·*a*) ⟷ *a* ≠ ⊥
a ≠ ⊥ ⟹ *eq*·*a*·*a* = *TT*
⟨*proof*⟩

lemma *eq-FF*[*simp*]:

$(FF = eq{\cdot}xs{\cdot}ys) \longleftrightarrow (xs \neq \bot \wedge ys \neq \bot \wedge xs \neq ys)$
$(eq{\cdot}xs{\cdot}ys = FF) \longleftrightarrow (xs \neq \bot \wedge ys \neq \bot \wedge xs \neq ys)$
⟨*proof*⟩

lemma *eq-TT*[*simp*]:
$(TT = eq{\cdot}xs{\cdot}ys) \longleftrightarrow (xs \neq \bot \wedge ys \neq \bot \wedge xs = ys)$
$(eq{\cdot}xs{\cdot}ys = TT) \longleftrightarrow (xs \neq \bot \wedge ys \neq \bot \wedge xs = ys)$
⟨*proof*⟩

end

instance *Integer* :: *Eq-def* ⟨*proof*⟩

2.3 Recursive let bindings

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

See §4.9 for an example use.

definition
CLetrec :: $('a{::}pcpo \to 'a \times 'b{::}pcpo) \to 'b$ **where**
$CLetrec = (\Lambda\ F.\ prod.snd\ (F{\cdot}(\mu\ x.\ prod.fst\ (F{\cdot}x))))$

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

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

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

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

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

3 Strict lists

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

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

lemma *scons-strict*[*simp*]: *scons*·⊥ = ⊥

⟨*proof*⟩

lemma *shead-bottom-iff*[*simp*]: (*shead*·*xs* = ⊥) ⟷ (*xs* = ⊥ ∨ *xs* = [::])
⟨*proof*⟩

lemma *stail-bottom-iff*[*simp*]: (*stail*·*xs* = ⊥) ⟷ (*xs* = ⊥ ∨ *xs* = [::])
⟨*proof*⟩

lemma *match-snil-match-scons-slist-case*: *match-snil*·*xs*·*k1* +++ *match-scons*·*xs*·*k2* = *slist-case*·*k1*·*k2*·*xs*
⟨*proof*⟩

lemma *slist-bottom'*: *slist-case*·⊥·⊥·*xs* = ⊥
⟨*proof*⟩

lemma *slist-bottom*[*simp*]: *slist-case*·⊥·⊥ = ⊥
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *slist-case-cong*:
assumes *xs* = *xs'*
assumes *xs'* = [::] ⟹ *n* = *n'*
assumes ⋀*y ys*. ⟦*xs'* = *y* :# *ys*; *y* ≠ ⊥; *ys* ≠ ⊥⟧ ⟹ *c y ys* = *c' y ys*
assumes *cont* (λ(*x*, *y*). *c x y*)
assumes *cont* (λ(*x*, *y*). *c' x y*)
shows *slist-case*·*n*·(Λ *x xs*. *c x xs*)·*xs* = *slist-case*·*n'*·(Λ *x xs*. *c' x xs*)·*xs'*
⟨*proof*⟩

Section syntax for *scons* ala Haskell.

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

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

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

Class instances.

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

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

| $[\![y \neq \bot;\ ys \neq \bot]\!] \Longrightarrow \mathit{eq\text{-}slist}{\cdot}[::]{\cdot}(y :\# ys) = \mathit{FF}$
| $[\![x \neq \bot;\ xs \neq \bot;\ y \neq \bot;\ ys \neq \bot]\!] \Longrightarrow \mathit{eq\text{-}slist}{\cdot}(x :\# xs){\cdot}(y :\# ys) = (\mathit{eq}{\cdot}x{\cdot}y\ \mathit{andalso}\ \mathit{eq\text{-}slist}{\cdot}xs{\cdot}ys)$

instance ⟨*proof*⟩

end

instance *slist* :: (*Eq-sym*) *Eq-sym*
⟨*proof*⟩

instance *slist* :: (*Eq-equiv*) *Eq-equiv*
⟨*proof*⟩

instance *slist* :: (*Eq-eq*) *Eq-eq*
⟨*proof*⟩

instance *slist* :: (*Eq-def*) *Eq-def*
⟨*proof*⟩

lemma *slist-eq-TT-snil*[*simp*]:
  fixes *xs* :: [:'*a*::*Eq*:]
  shows $(\mathit{eq}{\cdot}xs{\cdot}[::] = \mathit{TT}) \longleftrightarrow (xs = [::])$
    $(\mathit{eq}{\cdot}[::]{\cdot}xs = \mathit{TT}) \longleftrightarrow (xs = [::])$
⟨*proof*⟩

lemma *slist-eq-FF-snil*[*simp*]:
  fixes *xs* :: [:'*a*::*Eq*:]
  shows $(\mathit{eq}{\cdot}xs{\cdot}[::] = \mathit{FF}) \longleftrightarrow (\exists\, y\ ys.\ y \neq \bot \wedge ys \neq \bot \wedge xs = y :\# ys)$
    $(\mathit{eq}{\cdot}[::]{\cdot}xs = \mathit{FF}) \longleftrightarrow (\exists\, y\ ys.\ y \neq \bot \wedge ys \neq \bot \wedge xs = y :\# ys)$
⟨*proof*⟩

3.1 Some of the usual reasoning infrastructure

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

lemma *slistmem-bottom1*[*iff*]:
  fixes *x* :: $'a$
  shows $\neg\ \mathit{slistmem}\ x\ \bot$
⟨*proof*⟩

lemma *slistmem-bottom2*[*iff*]:
  fixes *xs* :: [:'*a*:]
  shows $\neg\ \mathit{slistmem}\ \bot\ xs$
⟨*proof*⟩

lemma *slistmem-nil*[*iff*]:
  shows $\neg\ \mathit{slistmem}\ x\ [::]$
⟨*proof*⟩

lemma *slistmem-scons*[*simp*]:
  shows $\mathit{slistmem}\ x\ (y :\# ys) \longleftrightarrow (x = y \wedge x \neq \bot \wedge ys \neq \bot) \vee (\mathit{slistmem}\ x\ ys \wedge y \neq \bot)$
⟨*proof*⟩

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

lemma *sset-simp*[*simp*]:
shows *sset* $\bot$ = {}
and *sset* [::] = {}
and $[\![x \neq \bot; xs \neq \bot]\!] \Longrightarrow$ *sset* (*x* :# *xs*) = *insert x* (*sset xs*)
⟨*proof*⟩

lemma *sset-defined*[*simp*]:
assumes $x \in$ *sset xs*
shows $x \neq \bot$
⟨*proof*⟩

lemma *sset-below*:
assumes $y \in$ *sset ys*
assumes $xs \sqsubseteq ys$
assumes $xs \neq \bot$
obtains *x* **where** $x \in$ *sset xs* **and** $x \sqsubseteq y$
⟨*proof*⟩

3.2 Some of the usual operations

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

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

lemma *snull-strict*[*simp*]: *snull*·$\bot$ = $\bot$
⟨*proof*⟩

lemma *snull-bottom-iff*[*simp*]: (*snull*·*xs* = $\bot$) ⟷ (*xs* = $\bot$)
⟨*proof*⟩

lemma *snull-FF-conv*: (*snull*·*xxs* = *FF*) ⟷ (∃ *x xs*. *xxs* ≠ $\bot$ ∧ *xxs* = *x* :# *xs*)
⟨*proof*⟩

lemma *snull-TT-conv*[*simp*]: (*snull*·*xs* = *TT*) ⟷ (*xs* = [::])
⟨*proof*⟩

lemma *snull-eq-snil*: *snull*·*xs* = *eq*·*xs*·[::]
⟨*proof*⟩

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

lemma *smap-strict*[*simp*]: *smap*·*f*·$\bot$ = $\bot$
⟨*proof*⟩

lemma *smap-bottom-iff*[*simp*]: (*smap*·*f*·*xs* = $\bot$) ⟷ (*xs* = $\bot$ ∨ (∃ *x*∈*sset xs*. *f*·*x* = $\bot$))
⟨*proof*⟩

lemma *smap-is-snil-conv*[*simp*]:
(*smap*·*f*·*xs* = [::]) ⟷ (*xs* = [::])
([::] = *smap*·*f*·*xs*) ⟷ (*xs* = [::])
⟨*proof*⟩

lemma *smap-strict-scons*[*simp*]:

assumes $f{\cdot}\bot = \bot$
shows $smap{\cdot}f{\cdot}(x :\# xs) = f{\cdot}x :\# smap{\cdot}f{\cdot}xs$
⟨*proof*⟩

lemma *smap-ID'*: $smap{\cdot}ID{\cdot}xs = xs$
⟨*proof*⟩

lemma *smap-ID*[*simp*]: $smap{\cdot}ID = ID$
⟨*proof*⟩

lemma *smap-cong*:
assumes $xs = xs'$
assumes $\bigwedge x.\ x \in sset\ xs \Longrightarrow f{\cdot}x = f'{\cdot}x$
shows $smap{\cdot}f{\cdot}xs = smap{\cdot}f'{\cdot}xs'$
⟨*proof*⟩

lemma *smap-smap'*[*simp*]:
assumes $f{\cdot}\bot = \bot$
shows $smap{\cdot}f{\cdot}(smap{\cdot}g{\cdot}xs) = smap{\cdot}(f\ oo\ g){\cdot}xs$
⟨*proof*⟩

lemma *smap-smap*[*simp*]:
assumes $f{\cdot}\bot = \bot$
shows $smap{\cdot}f\ oo\ smap{\cdot}g = smap{\cdot}(f\ oo\ g)$
⟨*proof*⟩

lemma *sset-smap*[*simp*]:
assumes $\bigwedge x.\ x \in sset\ xs \Longrightarrow f{\cdot}x \neq \bot$
shows $sset\ (smap{\cdot}f{\cdot}xs) = \{\ f{\cdot}x \mid x.\ x \in sset\ xs\ \}$
⟨*proof*⟩

lemma *shead-smap-distr*:
assumes $f{\cdot}\bot = \bot$
assumes $\bigwedge x.\ x{\in}sset\ xs \Longrightarrow f{\cdot}x \neq \bot$
shows $shead{\cdot}(smap{\cdot}f{\cdot}xs) = f{\cdot}(shead{\cdot}xs)$
⟨*proof*⟩

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

abbreviation *sappend-syn* :: *'a slist* ⇒ *'a slist* ⇒ *'a slist* (**infixr** ‹:@› *65*) **where**
$xs :@ ys \equiv sappend{\cdot}xs{\cdot}ys$

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

lemma *sappend-strict2*[*simp*]: $xs :@ \bot = \bot$
⟨*proof*⟩

lemma *sappend-bottom-iff*[*simp*]: $(xs :@ ys = \bot) \longleftrightarrow (xs = \bot \vee ys = \bot)$
⟨*proof*⟩

lemma *sappend-scons*[*simp*]: $(x :\# xs) :@ ys = x :\# xs :@ ys$
⟨*proof*⟩

lemma *sappend-assoc*[*simp*]: $(xs :@ ys) :@ zs = xs :@ (ys :@ zs)$
⟨*proof*⟩

lemma *sappend-snil-id-left*[*simp*]: $sappend{\cdot}[::] = ID$
⟨*proof*⟩

lemma *sappend-snil-id-right*[*iff*]: $xs :@ [::] = xs$
⟨*proof*⟩

lemma *snil-append-iff*[*iff*]: $xs :@ ys = [::] \longleftrightarrow xs = [::] \wedge ys = [::]$
⟨*proof*⟩

lemma *smap-sappend*[*simp*]: $smap{\cdot}f{\cdot}(xs :@ ys) = smap{\cdot}f{\cdot}xs :@ smap{\cdot}f{\cdot}ys$
⟨*proof*⟩

lemma *stail-sappend*: $stail{\cdot}(xs :@ ys) = (case\ xs\ of\ [::] \Rightarrow stail{\cdot}ys \mid z :\# zs \Rightarrow zs :@ ys)$
⟨*proof*⟩

lemma *stail-append2*[*simp*]: $xs \neq [::] \Longrightarrow stail{\cdot}(xs :@ ys) = stail{\cdot}xs :@ ys$
⟨*proof*⟩

lemma *slist-case-snoc*:
$g{\cdot}\bot{\cdot}\bot = \bot \Longrightarrow slist\text{-}case{\cdot}f{\cdot}g{\cdot}(xs :@ [:x:]) = g{\cdot}(shead{\cdot}(xs :@ [:x:])){\cdot}(stail{\cdot}(xs :@ [:x:]))$
⟨*proof*⟩

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

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

lemma *sall-const-TT*[*simp*]:
assumes $xs \neq \bot$
shows $sall{\cdot}(\Lambda\ x.\ TT){\cdot}xs = TT$
⟨*proof*⟩

lemma *sall-const-TT-conv*[*simp*]: $(sall{\cdot}(\Lambda\ x.\ TT){\cdot}xs = TT) \longleftrightarrow (xs \neq \bot)$
⟨*proof*⟩

lemma *sall-TT*[*simp*]: $(sall{\cdot}p{\cdot}xs = TT) \longleftrightarrow (xs \neq \bot \wedge (\forall\, x{\in}sset\ xs.\ p{\cdot}x = TT))$
⟨*proof*⟩

fixrec *sfilter* :: $('a \to tr) \to [:'a:] \to [:'a:]$ **where**
$sfilter{\cdot}p{\cdot}[::] = [::]$
| $[\![x \neq \bot;\ xs \neq \bot]\!] \Longrightarrow sfilter{\cdot}p{\cdot}(x :\# xs) = If\ p{\cdot}x\ then\ x :\# sfilter{\cdot}p{\cdot}xs\ else\ sfilter{\cdot}p{\cdot}xs$

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

lemma *sfilter-bottom-iff*[*simp*]: $(sfilter{\cdot}p{\cdot}xs = \bot) \longleftrightarrow (xs = \bot \vee (\exists\, x{\in}sset\ xs.\ p{\cdot}x = \bot))$
⟨*proof*⟩

lemma *sset-sfilter*[*simp*]:
assumes $\bigwedge x.\ x \in sset\ xs \Longrightarrow p{\cdot}x \neq \bot$
shows $sset\ (sfilter{\cdot}p{\cdot}xs) = \{x \mid x.\ x \in sset\ xs \wedge p{\cdot}x = TT\}$
⟨*proof*⟩

lemma *sfilter-strict-scons*[*simp*]:
assumes $p{\cdot}\bot = \bot$

shows *sfilter·p·(x :# xs) = If p·x then x :# sfilter·p·xs else sfilter·p·xs*
⟨*proof*⟩

lemma *sfilter-scons-let*:
assumes *p·⊥ = ⊥*
shows *sfilter·p·(x :# xs) = (let xs′ = sfilter·p·xs in If p·x then x :# xs′ else xs′)*
⟨*proof*⟩

lemma *sfilter-sappend*[*simp*]: *sfilter·p·(xs :@ ys) = sfilter·p·xs :@ sfilter·p·ys*
⟨*proof*⟩

lemma *sfilter-const-FF*[*simp*]:
assumes *xs ≠ ⊥*
shows *sfilter·(Λ x. FF)·xs = [::]*
⟨*proof*⟩

lemma *sfilter-const-FF-conv*[*simp*]: *(sfilter·(Λ x. FF)·xs = [::]) ⟷ (xs ≠ ⊥)*
⟨*proof*⟩

lemma *sfilter-const-TT*[*simp*]: *sfilter·(Λ x. TT)·xs = xs*
⟨*proof*⟩

lemma *sfilter-cong*:
assumes *xs = xs′*
assumes *⋀x. x ∈ sset xs ⟹ p·x = p′·x*
shows *sfilter·p·xs = sfilter·p′·xs′*
⟨*proof*⟩

lemma *sfilter-snil-conv*[*simp*]: *sfilter·p·xs = [::] ⟷ sall·(neg oo p)·xs = TT*
⟨*proof*⟩

lemma *sfilter-sfilter′*: *sfilter·p·(sfilter·q·xs) = sfilter·(Λ x. q·x andalso p·x)·xs*
⟨*proof*⟩

lemma *sfilter-sfilter*: *sfilter·p oo sfilter·q = sfilter·(Λ x. q·x andalso p·x)*
⟨*proof*⟩

lemma *sfilter-smap′*:
assumes *p·⊥ = ⊥*
shows *sfilter·p·(smap·f·xs) = smap·f·(sfilter·(p oo f)·xs)*
⟨*proof*⟩

lemma *sfilter-smap*:
assumes *p·⊥ = ⊥*
shows *sfilter·p oo smap·f = smap·f oo sfilter·(p oo f)*
⟨*proof*⟩

fixrec *sfoldl* :: *('a::pcpo → 'b::domain → 'a) → 'a → [:'b:] → 'a* **where**
sfoldl·f·z·[::] = z
| *⟦x ≠ ⊥; xs ≠ ⊥⟧ ⟹ sfoldl·f·z·(x :# xs) = sfoldl·f·(f·z·x)·xs*

lemma *sfoldl-strict*[*simp*]: *sfoldl·f·z·⊥ = ⊥*
⟨*proof*⟩

lemma *sfoldl-strict-f*[*simp*]:
assumes *f·⊥ = ⊥*
shows *sfoldl·f·⊥·xs = ⊥*
⟨*proof*⟩

lemma *sfoldl-cong*:
  assumes $xs = xs'$
  assumes $z = z'$
  assumes $\bigwedge x\ z.\ x \in sset\ xs \Longrightarrow f{\cdot}z{\cdot}x = f'{\cdot}z{\cdot}x$
  shows $sfoldl{\cdot}f{\cdot}z{\cdot}xs = sfoldl{\cdot}f'{\cdot}z'{\cdot}xs'$
⟨*proof*⟩

lemma *sfoldl-sappend*[*simp*]:
  assumes $f{\cdot}\bot = \bot$
  shows $sfoldl{\cdot}f{\cdot}z{\cdot}(xs :@ ys) = sfoldl{\cdot}f{\cdot}(sfoldl{\cdot}f{\cdot}z{\cdot}xs){\cdot}ys$
⟨*proof*⟩

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

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

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

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

lemma *sconcat-scons*[*simp*]:
  shows $sconcat{\cdot}(x :\# xs) = x :@ sconcat{\cdot}xs$
⟨*proof*⟩

lemma *sconcat-sfoldl-aux*: $sfoldl{\cdot}sappend{\cdot}z{\cdot}xs = z :@ sconcat{\cdot}xs$
⟨*proof*⟩

lemma *sconcat-sfoldl*: $sconcat = sfoldl{\cdot}sappend{\cdot}[::]$
⟨*proof*⟩

lemma *sconcat-sappend*[*simp*]: $sconcat{\cdot}(xs :@ ys) = sconcat{\cdot}xs :@ sconcat{\cdot}ys$
⟨*proof*⟩

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

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

lemma *slength-bottom-iff*[*simp*]: $(slength{\cdot}xs = \bot) \longleftrightarrow (xs = \bot)$
⟨*proof*⟩

lemma *slength-ge-0*: $slength{\cdot}xs = MkI{\cdot}n \Longrightarrow n \geq 0$
⟨*proof*⟩

lemma *slengthE*:
  shows $[\![xs \neq \bot;\ \bigwedge n.\ [\![slength{\cdot}xs = MkI{\cdot}n;\ 0 \leq n]\!] \Longrightarrow Q]\!] \Longrightarrow Q$
⟨*proof*⟩

lemma *slength-0-conv*[*simp*]:
  $(slength{\cdot}xs = 0) \longleftrightarrow (xs = [::])$
  $(slength{\cdot}xs = MkI{\cdot}0) \longleftrightarrow (xs = [::])$
  $eq{\cdot}0{\cdot}(slength{\cdot}xs) = snull{\cdot}xs$
  $eq{\cdot}(slength{\cdot}xs){\cdot}0 = snull{\cdot}xs$
⟨*proof*⟩

lemma *le-slength-0*[*simp*]: $(le{\cdot}0{\cdot}(slength{\cdot}xs) = TT) \longleftrightarrow (xs \neq \bot)$
⟨*proof*⟩

lemma *lt-slength-0*[*simp*]:
  $xs \neq \bot \Longrightarrow lt{\cdot}(slength{\cdot}xs){\cdot}0 = FF$
  $xs \neq \bot \Longrightarrow lt{\cdot}(slength{\cdot}xs){\cdot}(slength{\cdot}xs + 1) = TT$
⟨*proof*⟩

lemma *slength-smap*[*simp*]:
  assumes $\bigwedge x.\ x \neq \bot \Longrightarrow f{\cdot}x \neq \bot$
  shows $slength{\cdot}(smap{\cdot}f{\cdot}xs) = slength{\cdot}xs$
⟨*proof*⟩

lemma *slength-sappend*[*simp*]: $slength{\cdot}(xs :@ ys) = slength{\cdot}xs + slength{\cdot}ys$
⟨*proof*⟩

lemma *slength-sfoldl-aux*: $sfoldl{\cdot}(\Lambda\ i\ {-}.\ i + 1){\cdot}z{\cdot}xs = z + slength{\cdot}xs$
⟨*proof*⟩

lemma *slength-sfoldl*: $slength = sfoldl{\cdot}(\Lambda\ i\ {-}.\ i + 1){\cdot}0$
⟨*proof*⟩

lemma *le-slength-plus*:
  assumes $xs \neq \bot$
  assumes $n \neq \bot$
  shows $le{\cdot}n{\cdot}(slength{\cdot}xs + n) = TT$
⟨*proof*⟩

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

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

lemma *srev-bottom-iff*[*simp*]: $(srev{\cdot}xs = \bot) \longleftrightarrow (xs = \bot)$
⟨*proof*⟩

lemma *srev-scons*[*simp*]: $srev{\cdot}(x :\# xs) = srev{\cdot}xs :@ [:x:]$
⟨*proof*⟩

lemma *srev-sappend*[*simp*]: $srev{\cdot}(xs :@ ys) = srev{\cdot}ys :@ srev{\cdot}xs$
⟨*proof*⟩

lemma *srev-srev-ident*[*simp*]: $srev{\cdot}(srev{\cdot}xs) = xs$
⟨*proof*⟩

lemma *srev-cases*[*case-names bottom snil ssnoc*]:
  assumes $xs = \bot \Longrightarrow P$
  assumes $xs = [::] \Longrightarrow P$
  assumes $\bigwedge y\ ys.\ [\![y \neq \bot;\ ys \neq \bot;\ xs = ys :@ [:y:]]\!] \Longrightarrow P$

shows P
⟨*proof*⟩

lemma *srev-induct*[*case-names bottom snil ssnoc*]:
assumes $P\ \bot$
assumes $P\ [::]$
assumes $\bigwedge x\ xs.\ [\![x \neq \bot;\ xs \neq \bot;\ P\ xs]\!] \Longrightarrow P\ (xs\ :@\ [:x:])$
shows $P\ xs$
⟨*proof*⟩

lemma *sfoldr-conv-sfoldl*:
assumes $\bigwedge x.\ f{\cdot}x{\cdot}\bot = \bot$ — f must be strict in the accumulator.
shows $sfoldr{\cdot}f{\cdot}z{\cdot}xs = sfoldl{\cdot}(\Lambda\ acc\ x.\ f{\cdot}x{\cdot}acc){\cdot}z{\cdot}(srev{\cdot}xs)$
⟨*proof*⟩

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

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

lemma *stake-bottom-iff*[*simp*]: $(stake{\cdot}i{\cdot}xs = \bot) \longleftrightarrow (i = \bot \vee xs = \bot)$
⟨*proof*⟩

lemma *stake-0*[*simp*]:
$xs \neq \bot \Longrightarrow stake{\cdot}0{\cdot}xs = [::]$
$xs \neq \bot \Longrightarrow stake{\cdot}(MkI{\cdot}0){\cdot}xs = [::]$
$stake{\cdot}0{\cdot}xs \sqsubseteq [::]$
⟨*proof*⟩

lemma *stake-scons*[*simp*]: $le{\cdot}1{\cdot}i = TT \Longrightarrow stake{\cdot}i{\cdot}(x\ :\#\ xs) = x\ :\#\ stake{\cdot}(i - 1){\cdot}xs$
⟨*proof*⟩

lemma *take-MkI-scons*[*simp*]:
$0 < n \Longrightarrow stake{\cdot}(MkI{\cdot}n){\cdot}(x\ :\#\ xs) = x\ :\#\ stake{\cdot}(MkI{\cdot}(n - 1)){\cdot}xs$
⟨*proof*⟩

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

lemma *stake-all*:
assumes $le{\cdot}(slength{\cdot}xs){\cdot}i = TT$
shows $stake{\cdot}i{\cdot}xs = xs$
⟨*proof*⟩

lemma *stake-all-triv*[*simp*]: $stake{\cdot}(slength{\cdot}xs){\cdot}xs = xs$
⟨*proof*⟩

lemma *stake-append*[*simp*]: $stake{\cdot}i{\cdot}(xs\ :@\ ys) = stake{\cdot}i{\cdot}xs\ :@\ stake{\cdot}(i - slength{\cdot}xs){\cdot}ys$
⟨*proof*⟩

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*·⊥ = ⊥
⟨*proof*⟩

lemma *sdrop-bottom-iff*[*simp*]: (*sdrop*·*i*·*xs* = ⊥) ⟷ (*i* = ⊥ ∨ *xs* = ⊥)
⟨*proof*⟩

lemma *sdrop-snil*[*simp*]:
  assumes *i* ≠ ⊥
  shows *sdrop*·*i*·[::] = [::]
⟨*proof*⟩

lemma *sdrop-snil-conv*[*simp*]: (*sdrop*·*i*·[::] = [::]) ⟷ (*i* ≠ ⊥)
⟨*proof*⟩

lemma *sdrop-0*[*simp*]:
  sdrop·*0*·*xs* = *xs*
  sdrop·(*MkI*·*0*)·*xs* = *xs*
⟨*proof*⟩

lemma *sdrop-pos*:
  le·*i*·*0* = *FF* ⟹ *sdrop*·*i*·*xs* = (*case xs of* [::] ⇒ [::] | *y* :# *ys* ⇒ *sdrop*·(*i* − *1*)·*ys*)
⟨*proof*⟩

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

lemma *sdrop-numeral-scons*[*simp*]:
  x ≠ ⊥ ⟹ *sdrop*·*1*·(*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*
⟨*proof*⟩

lemma *sdrop-sappend*[*simp*]:
  sdrop·*i*·(*xs* :@ *ys*) = *sdrop*·*i*·*xs* :@ *sdrop*·(*i* − *slength*·*xs*)·*ys*
⟨*proof*⟩

lemma *sdrop-all*:
  assumes *le*·(*slength*·*xs*)·*i* = *TT*
  shows *sdrop*·*i*·*xs* = [::]
⟨*proof*⟩

lemma *slength-sdrop*[*simp*]:
  slength·(*sdrop*·*i*·*xs*) = *If le*·*i*·*0 then slength*·*xs else If le*·(*slength*·*xs*)·*i then 0 else slength*·*xs* − *i*
⟨*proof*⟩

lemma *sdrop-not-snilD*:
  assumes *sdrop*·(*MkI*·*i*)·*xs* ≠ [::]
  assumes *xs* ≠ ⊥
  shows *lt*·(*MkI*·*i*)·(*slength*·*xs*) = *TT* ∧ *xs* ≠ [::]
⟨*proof*⟩

lemma *sdrop-sappend-same*:

assumes $xs \neq \bot$
shows $sdrop{\cdot}(slength{\cdot}xs){\cdot}(xs :@ ys) = ys$
⟨*proof*⟩

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

lemma *sscanl-strict*[*simp*]:
$sscanl{\cdot}f{\cdot}\bot{\cdot}xs = \bot$
$sscanl{\cdot}f{\cdot}z{\cdot}\bot = \bot$
⟨*proof*⟩

lemma *sscanl-cong*:
assumes $xs = xs'$
assumes $z = z'$
assumes $\bigwedge x\ z.\ x \in sset\ xs \Longrightarrow f{\cdot}z{\cdot}x = f'{\cdot}z{\cdot}x$
shows $sscanl{\cdot}f{\cdot}z{\cdot}xs = sscanl{\cdot}f'{\cdot}z'{\cdot}xs'$
⟨*proof*⟩

lemma *sscanl-lfp-fusion'*:
assumes $g{\cdot}\bot = \bot$
assumes $*$: $\bigwedge acc\ x.\ x \neq \bot \Longrightarrow g{\cdot}(f{\cdot}acc{\cdot}x) = f'{\cdot}(g{\cdot}acc){\cdot}x$
shows $smap{\cdot}g{\cdot}(sscanl{\cdot}f{\cdot}z{\cdot}xs) = sscanl{\cdot}f'{\cdot}(g{\cdot}z){\cdot}xs$
⟨*proof*⟩

lemma *sscanl-lfp-fusion*:
assumes $g{\cdot}\bot = \bot$
assumes $*$: $\bigwedge acc\ x.\ x \neq \bot \Longrightarrow g{\cdot}(f{\cdot}acc{\cdot}x) = f'{\cdot}(g{\cdot}acc){\cdot}x$
shows $smap{\cdot}g\ oo\ sscanl{\cdot}f{\cdot}z = sscanl{\cdot}f'{\cdot}(g{\cdot}z)$
⟨*proof*⟩

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

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

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

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

lemma *sinits-bottom-iff*[*simp*]: (*sinits*·*xs* = ⊥) ⟷ (*xs* = ⊥)
⟨*proof*⟩

lemma *sinits-not-snil*[*iff*]: *sinits*·*xs* ≠ [::]
⟨*proof*⟩

lemma *sinits-empty-bottom*[*simp*]: (*sset* (*sinits*·*xs*) = {}) ⟷ (*xs* = ⊥)
⟨*proof*⟩

lemma *sinits-scons*[*simp*]: *sinits*·(*x* :# *xs*) = [::] :# *smap*·(*x* :#)·(*sinits*·*xs*)
⟨*proof*⟩

lemma *sinits-length*[*simp*]: *slength*·(*sinits*·*xs*) = *slength*·*xs* + *1*
⟨*proof*⟩

lemma *sinits-snoc*[*simp*]: *sinits*·(*xs* :@ [:*x*:]) = *sinits*·*xs* :@ [:*xs* :@ [:*x*:]:]
⟨*proof*⟩

lemma *sinits-foldr′*: — Bird (1987, p30)
shows *sinits*·*xs* = *sfoldr*·(Λ *x xs*. [:[::]:] :@ *smap*·(*x* :#)·*xs*)·[:[::]:]·*xs*
⟨*proof*⟩

lemma *sinits-sscanl′*:
shows *smap*·(*sfoldl*·*f*·*z*)·(*sinits*·*xs*) = *sscanl*·*f*·*z*·*xs*
⟨*proof*⟩

lemma *sinits-sscanl*: — Bird (1987, Lemma 5), Bird (2010, p118 “the scan lemma”)
shows *smap*·(*sfoldl*·*f*·*z*) *oo sinits* = *sscanl*·*f*·*z*
⟨*proof*⟩

lemma *sinits-all*[*simp*]: (*xs* ∈ *sset* (*sinits*·*xs*)) ⟷ (*xs* ≠ ⊥)
⟨*proof*⟩

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

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

lemma *stails-bottom-iff*[*simp*]: (*stails*·*xs* = ⊥) ⟷ (*xs* = ⊥)
⟨*proof*⟩

lemma *stails-not-snil*[*iff*]: *stails*·*xs* ≠ [::]
⟨*proof*⟩

lemma *stails-scons*[*simp*]: *stails*·(*x* :# *xs*) = (*x* :# *xs*) :# *stails*·*xs*
⟨*proof*⟩

lemma *stails-slength*[*simp*]: *slength*·(*stails*·*xs*) = *slength*·*xs* + *1*
⟨*proof*⟩

lemma *stails-snoc*[*simp*]:
shows *stails*·(*xs* :@ [:*x*:]) = *smap*·(Λ *ys*. *ys* :@ [:*x*:])·(*stails*·*xs*) :@ [:[::]:]
⟨*proof*⟩

lemma *stails-sfoldl'*:
shows $stails{\cdot}xs = sfoldl{\cdot}(\Lambda\ xs\ x.\ smap{\cdot}(\Lambda\ ys.\ ys :@ [:x:]){\cdot}xs :@ [:[::]:]){\cdot}[:[::]:]{\cdot}xs$
⟨*proof*⟩

lemma *stails-sfoldl*:
shows $stails = sfoldl{\cdot}(\Lambda\ xs\ x.\ smap{\cdot}(\Lambda\ ys.\ ys :@ [:x:]){\cdot}xs :@ [:[::]:]){\cdot}[:[::]:]$
⟨*proof*⟩

lemma *stails-all*[*simp*]: $(xs \in sset\ (stails{\cdot}xs)) \longleftrightarrow (xs \neq \bot)$
⟨*proof*⟩

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

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

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

lemma *selem-sappend*[*simp*]:
assumes $ys \neq \bot$
shows $selem{\cdot}x{\cdot}(xs :@ ys) = (selem{\cdot}x{\cdot}xs\ orelse\ selem{\cdot}x{\cdot}ys)$
⟨*proof*⟩

lemma *elem-TT*[*simp*]: $(selem{\cdot}x{\cdot}xs = TT) \longleftrightarrow (x \in sset\ xs)$
⟨*proof*⟩

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

lemma *selem-snil-stails*[*iff*]:
assumes $xs \neq \bot$
shows $selem{\cdot}[::]{\cdot}(stails{\cdot}xs) = TT$
⟨*proof*⟩

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

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

lemma *sconcatMap-snil*[*simp*]: $sconcatMap{\cdot}f{\cdot}[::] = [::]$
⟨*proof*⟩

lemma *sconcatMap-scons*[*simp*]: $x \neq \bot \Longrightarrow sconcatMap{\cdot}f{\cdot}(x :\# xs) = f{\cdot}x :@ sconcatMap{\cdot}f{\cdot}xs$
⟨*proof*⟩

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

lemma *sconcatMap-sappend*[*simp*]: $sconcatMap{\cdot}f{\cdot}(xs :@ ys) = sconcatMap{\cdot}f{\cdot}xs :@ sconcatMap{\cdot}f{\cdot}ys$
⟨*proof*⟩

lemma *sconcatMap-monad-laws*:
$sconcatMap{\cdot}(\Lambda\ x.\ [:x:]){\cdot}xs = xs$

sconcatMap·g·(sconcatMap·f·xs) = sconcatMap·(Λ x. sconcatMap·g·(f·x))·xs
⟨*proof*⟩

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

lemma *upto-strict*[*simp*]:
supto·⊥ = ⊥
supto·m·⊥ = ⊥
⟨*proof*⟩

lemma *supto-is-snil-conv*[*simp*]:
(supto·(MkI·i)·(MkI·j) = [::]) ⟷ (j < i)
([::] = supto·(MkI·i)·(MkI·j)) ⟷ (j < i)
⟨*proof*⟩

lemma *supto-simp*[*simp*]:
j < i ⟹ supto·(MkI·i)·(MkI·j) = [::]
i ≤ j ⟹ supto·(MkI·i)·(MkI·j) = MkI·i :# supto·(MkI·i+1)·(MkI·j)
supto·0·0 = [:0:]
⟨*proof*⟩

lemma *supto-defined*[*simp*]: *supto·(MkI·i)·(MkI·j) ≠ ⊥* (**is** *?P i j*)
⟨*proof*⟩

lemma *supto-bottom-iff*[*simp*]:
(supto·i·j = ⊥) ⟷ (i = ⊥ ∨ j = ⊥)
⟨*proof*⟩

lemma *supto-snoc*[*simp*]:
i ≤ j ⟹ supto·(MkI·i)·(MkI·j) = supto·(MkI·i)·(MkI·j−1) :@ [:*MkI·j*:]
⟨*proof*⟩

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*⟩

lemma *sset-supto*[*simp*]:
sset (supto·(MkI·i)·(MkI·j)) = {MkI·k |k. i ≤ k ∧ k ≤ j} (**is** *sset (?u i j) = ?R i j*)
⟨*proof*⟩

lemma *supto-split1*: — From *HOL.List*
assumes *i ≤ j*
assumes *j ≤ k*
shows *supto·(MkI·i)·(MkI·k) = supto·(MkI·i)·(MkI·(j − 1)) :@ supto·(MkI·j)·(MkI·k)*
⟨*proof*⟩

lemma *supto-split2*: — From *HOL.List*
assumes *i ≤ j*
assumes *j ≤ k*
shows *supto·(MkI·i)·(MkI·k) = supto·(MkI·i)·(MkI·j) :@ supto·(MkI·(j + 1))·(MkI·k)*
⟨*proof*⟩

lemma *supto-split3*: — From *HOL.List*
assumes *i ≤ j*
assumes *j ≤ k*
shows *supto·(MkI·i)·(MkI·k) = supto·(MkI·i)·(MkI·(j − 1)) :@ MkI·j :# supto·(MkI·(j + 1))·(MkI·k)*
⟨*proof*⟩

lemma *sinits-stake'*:
shows $sinits{\cdot}xs = smap{\cdot}(\Lambda\ i.\ stake{\cdot}i{\cdot}xs){\cdot}(supto{\cdot}0{\cdot}(slength{\cdot}xs))$
⟨*proof*⟩

lemma *stails-sdrop'*:
shows $stails{\cdot}xs = smap{\cdot}(\Lambda\ i.\ sdrop{\cdot}i{\cdot}xs){\cdot}(supto{\cdot}0{\cdot}(slength{\cdot}xs))$
⟨*proof*⟩

lemma *sdrop-elem-stails*[*iff*]:
assumes $xs \neq \bot$
shows $sdrop{\cdot}(MkI{\cdot}i){\cdot}xs \in sset\ (stails{\cdot}xs)$
⟨*proof*⟩

fixrec $slast :: [:'a:] \to 'a$ **where**
$slast{\cdot}[::] = \bot$
| $[\![x \neq \bot;\ xs \neq \bot]\!] \Longrightarrow slast{\cdot}(x :\# xs) = (case\ xs\ of\ [::] \Rightarrow x \mid y :\# ys \Rightarrow slast{\cdot}xs)$

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

lemma *slast-singleton*[*simp*]: $slast{\cdot}[:x:] = x$
⟨*proof*⟩

lemma *slast-sappend-ssnoc*[*simp*]:
assumes $xs \neq \bot$
shows $slast{\cdot}(xs :@ [:x:]) = x$
⟨*proof*⟩

fixrec $sbutlast :: [:'a:] \to [:'a:]$ **where**
$sbutlast{\cdot}[::] = [::]$
| $[\![x \neq \bot;\ xs \neq \bot]\!] \Longrightarrow sbutlast{\cdot}(x :\# xs) = (case\ xs\ of\ [::] \Rightarrow [::] \mid y :\# ys \Rightarrow x :\# sbutlast{\cdot}xs)$

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

lemma *sbutlast-sappend-ssnoc*[*simp*]:
assumes $x \neq \bot$
shows $sbutlast{\cdot}(xs :@ [:x:]) = xs$
⟨*proof*⟩

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

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

lemma *prefix-bottom-iff*[*simp*]: $(prefix{\cdot}xs{\cdot}ys = \bot) \longleftrightarrow (xs = \bot \vee ys = \bot)$
⟨*proof*⟩

lemma *prefix-definedD*:
assumes $prefix{\cdot}xs{\cdot}ys = TT$
shows $xs \neq \bot \wedge ys \neq \bot$
⟨*proof*⟩

lemma *prefix-refl*[*simp*]:
  assumes $xs \neq \bot$
  shows $prefix{\cdot}xs{\cdot}xs = TT$
⟨*proof*⟩

lemma *prefix-refl-conv*[*simp*]: $(prefix{\cdot}xs{\cdot}xs = TT) \longleftrightarrow (xs \neq \bot)$
⟨*proof*⟩

lemma *prefix-of-snil*[*simp*]: $prefix{\cdot}xs{\cdot}[::] = (case\ xs\ of\ [::] \Rightarrow TT \mid x :\# xs \Rightarrow FF)$
⟨*proof*⟩

lemma *prefix-singleton-TT*:
  shows $prefix{\cdot}[:x:]{\cdot}ys = TT \longleftrightarrow (x \neq \bot \wedge (\exists\, zs.\ zs \neq \bot \wedge ys = x :\# zs))$
⟨*proof*⟩

lemma *prefix-singleton-FF*:
  shows $prefix{\cdot}[:x:]{\cdot}ys = FF \longleftrightarrow (x \neq \bot \wedge (ys = [::] \vee (\exists\, z\ zs.\ z \neq \bot \wedge zs \neq \bot \wedge ys = z :\# zs \wedge x \neq z)))$
⟨*proof*⟩

lemma *prefix-FF-not-snilD*:
  assumes $prefix{\cdot}xs{\cdot}ys = FF$
  shows $xs \neq [::]$
⟨*proof*⟩

lemma *prefix-slength*:
  assumes $prefix{\cdot}xs{\cdot}ys = TT$
  shows $le{\cdot}(slength{\cdot}xs){\cdot}(slength{\cdot}ys) = TT$
⟨*proof*⟩

lemma *prefix-slength-strengthen*: $prefix{\cdot}xs{\cdot}ys = (le{\cdot}(slength{\cdot}xs){\cdot}(slength{\cdot}ys)\ andalso\ prefix{\cdot}xs{\cdot}ys)$
⟨*proof*⟩

lemma *prefix-scons-snil*[*simp*]: $prefix{\cdot}(x :\# xs){\cdot}[::] \neq TT$
⟨*proof*⟩

lemma *scons-prefix-scons*[*simp*]:
  $(prefix{\cdot}(x :\# xs){\cdot}(y :\# ys) = TT) \longleftrightarrow (eq{\cdot}x{\cdot}y = TT \wedge prefix{\cdot}xs{\cdot}ys = TT)$
⟨*proof*⟩

lemma *append-prefixD*:
  assumes $prefix{\cdot}(xs :@ ys){\cdot}zs = TT$
  shows $prefix{\cdot}xs{\cdot}zs = TT$
⟨*proof*⟩

lemma *same-prefix-prefix*[*simp*]:
  assumes $xs \neq \bot$
  shows $prefix{\cdot}(xs :@ ys){\cdot}(xs :@ zs) = prefix{\cdot}ys{\cdot}zs$
⟨*proof*⟩

lemma *eq-prefix-TT*:
  assumes $eq{\cdot}xs{\cdot}ys = TT$
  shows $prefix{\cdot}xs{\cdot}ys = TT$
⟨*proof*⟩

lemma *prefix-eq-FF*:
  assumes $prefix{\cdot}xs{\cdot}ys = FF$
  shows $eq{\cdot}xs{\cdot}ys = FF$

⟨*proof*⟩

lemma *prefix-slength-eq*:
shows *eq·xs·ys* = (*eq·*(*slength·xs*)*·*(*slength·ys*) *andalso prefix·xs·ys*)
⟨*proof*⟩

lemma *stake-slength-plus-1*:
shows *stake·*(*slength·xs* + *1*)*·*(*y* :# *ys*) = *y* :# *stake·*(*slength·xs*)*·ys*
⟨*proof*⟩

lemma *sdrop-slength-plus-1*:
assumes *y* ≠ ⊥
shows *sdrop·*(*slength·xs* + *1*)*·*(*y* :# *ys*) = *sdrop·*(*slength·xs*)*·ys*
⟨*proof*⟩

lemma *eq-take-length-prefix*: *prefix·xs·ys* = *eq·xs·*(*stake·*(*slength·xs*)*·ys*)
⟨*proof*⟩

lemma *prefix-sdrop-slength*:
assumes *prefix·xs·ys* = *TT*
shows *xs* :@ *sdrop·*(*slength·xs*)*·ys* = *ys*
⟨*proof*⟩

lemma *prefix-sdrop-prefix-eq*:
assumes *prefix·xs·ys* = *TT*
shows *eq·*(*sdrop·*(*slength·xs*)*·ys*)*·*[::] = *eq·ys·xs*
⟨*proof*⟩

4 Knuth-Morris-Pratt matching according to Bird

4.1 Step 1: Specification

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

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

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

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

lemma *matches·*[::]*·*[::] = [:*0*:]
⟨*proof*⟩

lemma *matches·*[::]*·*[:*10*::*Integer*, *20*, *30*:] = [:*0*, *1*, *2*, *3*:]
⟨*proof*⟩

lemma *matches·*[:*1*::*Integer*,*2*,*3*,*1*,*2*:]*·*[:*1*,*2*,*1*,*2*,*3*,*1*,*2*,*3*,*1*,*2*:] = [:*7*, *10*:]
⟨*proof*⟩

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

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

matches·⊥ = ⊥
matches·*pat*·⊥ = ⊥
⟨*proof*⟩

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

lemma *fork-sfoldl*:
shows *sfoldl*·*f1*·*z1* &&& *sfoldl*·*f2*·*z2* = *sfoldl*·(Λ (*a*, *b*) *z*. (*f1*·*a*·*z*, *f2*·*b*·*z*))·(*z1*, *z2*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *smap-sfilter-split-cfcomp*: — Bird (16.4)
assumes *f*·⊥ = ⊥
assumes *p*·⊥ = ⊥
shows *smap*·*f* *oo* *sfilter*·(*p* *oo* *g*) = *smap*·*cfst* *oo* *sfilter*·(*p* *oo* *csnd*) *oo* *smap*·(*f* &&& *g*) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *Bird-strategy*:
assumes *endswith*: *endswith*·*pat* = *p* *oo* *sfoldl*·*op*·*z*
assumes *step*: *step* = (Λ (*n*, *x*) *y*. (*n* + *1*, *op*·*x*·*y*))
assumes *p*·⊥ = ⊥ — We can reasonably expect the predicate to be strict
shows *matches*·*pat* = *smap*·*cfst* *oo* *sfilter*·(*p* *oo* *csnd*) *oo* *sscanl*·*step*·(*0*, *z*)
⟨*proof*⟩

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

lemma *sfilter-supto*:
assumes *0* ≤ *d*
shows *sfilter*·(Λ *x*. *le*·(*MkI*·*n* − *x*)·(*MkI*·*d*))·(*supto*·(*MkI*·*m*)·(*MkI*·*n*))
= *supto*·(*MkI*·(*if* *m* ≤ *n* − *d* *then* *n* − *d* *else* *m*))·(*MkI*·*n*) (**is** *?sfilterp*·*?suptomn* = -)
⟨*proof*⟩

lemma *endswith-eq-sdrop*: *endswith*·*pat*·*xs* = *eq*·*pat*·(*sdrop*·(*slength*·*xs* − *slength*·*pat*)·*xs*)
⟨*proof*⟩

lemma *endswith-def2*: — Bird p127
shows *endswith*·*pat*·*xs* = *eq*·*pat*·(*shead*·(*sfilter*·(Λ *x*. *prefix*·*x*·*pat*)·(*stails*·*xs*))) (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

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

lemma *split-strict*[*simp*]:
shows *split*·⊥ = ⊥
and *split*·*pat*·⊥ = ⊥
⟨*proof*⟩

lemma *split-bottom-iff*[*simp*]: (*split*·*pat*·*xs* = ⊥) ⟷ (*pat* = ⊥ ∨ *xs* = ⊥)
⟨*proof*⟩

lemma *split-snil*[*simp*]:
assumes *pat* ≠ ⊥
shows *split*·*pat*·[::] = ([::], *pat*)
⟨*proof*⟩

lemma *split-pattern*: — Bird p128, observation
assumes *xs* ≠ ⊥

assumes *split·pat·xs* = (*us*, *vs*)
shows *us* :@ *vs* = *pat*
⟨*proof*⟩

lemma *endswith-split*: — Bird p128, after defining *split*
shows *endswith·pat* = *snull oo csnd oo split·pat*
⟨*proof*⟩

lemma *split-length-lt*:
assumes *pat* ≠ ⊥
assumes *xs* ≠ ⊥
shows *lt·*(*slength·*(*prod.fst* (*split·pat·xs*)))·(*slength·xs* + *1*) = *TT*
⟨*proof*⟩

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

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

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

lemma *split-snoc*: — Bird p128
shows *split·pat·*(*xs* :@ [:*x*:]) = *split·pat·*(*cfst·*(*split·pat·xs*) :@ [:*x*:])
⟨*proof*⟩

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

lemma *op-strict*[*simp*]:
op·pat·⊥ = ⊥
op·pat·(*us*, ⊥) = ⊥
op·pat·usvs·⊥ = ⊥
⟨*proof*⟩

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

lemma *op-induct*[*case-names step*]:
fixes *usvs*:: [:'*a*:] × '*b*
assumes *step*: ⋀*usvs*. (⋀*usvs'*. *lt·*(*slength·*(*cfst·usvs'*))·(*slength·*(*cfst·usvs*)) = *TT* ⟹ *P usvs'*) ⟹ *P usvs*
shows *P usvs*
⟨*proof*⟩

lemma *op-induct'*[*case-names step*]:
assumes *step*: ⋀*us*. (⋀*us'*. *lt·*(*slength·us'*)·(*slength·us*) = *TT* ⟹ *P us'*) ⟹ *P us*
shows *P us*
⟨*proof*⟩

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

⟨*proof*⟩

lemma *split-sfoldl-op*:
assumes $pat \neq \bot$
shows $sfoldl{\cdot}(op{\cdot}pat){\cdot}([::],\ pat) = split{\cdot}pat$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

lemma *matches-op*:
shows $matches{\cdot}pat = smap{\cdot}cfst\ oo\ sfilter{\cdot}(snull\ oo\ csnd\ oo\ csnd)$
$oo\ sscanl{\cdot}(\Lambda\ (n,\ usvs)\ x.\ (n + 1,\ op{\cdot}pat{\cdot}usvs{\cdot}x)){\cdot}(0,\ ([::],\ pat))$ (**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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

fixrec
$op' :: [:'a::Eq\text{-}def:] \to [:'a:] \times [:'a:] \to 'a \to [:'a:] \times [:'a:]$
where
[*simp del*]:
$op'{\cdot}pat{\cdot}(us,\ vs){\cdot}x =$
(*If* $prefix{\cdot}[:x:]{\cdot}vs$ *then* $(us :@ [:x:],\ stail{\cdot}vs)$ — continue matching
else If $snull{\cdot}us$ *then* $([::],\ pat)$ — fail at the start of the pattern: discard x
else $sfoldl{\cdot}(op'{\cdot}pat){\cdot}([::],\ pat){\cdot}(stail{\cdot}us :@ [:x:])$ — fail later: discard $shead{\cdot}us$ and determine where to restart
)

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

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

lemma $op'\text{-}strict[simp]$:
$op'{\cdot}pat{\cdot}\bot = \bot$
$op'{\cdot}pat{\cdot}(us,\ \bot) = \bot$
$op'{\cdot}pat{\cdot}usvs{\cdot}\bot = \bot$
⟨*proof*⟩

lemma $sfoldl\text{-}op'\text{-}strict[simp]$:
$op'{\cdot}pat{\cdot}(sfoldl{\cdot}(op'{\cdot}pat){\cdot}(us,\ \bot){\cdot}xs){\cdot}x = \bot$
⟨*proof*⟩

lemma $op'\text{-}op$:
shows $op'{\cdot}pat{\cdot}usvs{\cdot}x = op{\cdot}pat{\cdot}usvs{\cdot}x$
⟨*proof*⟩

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

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

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

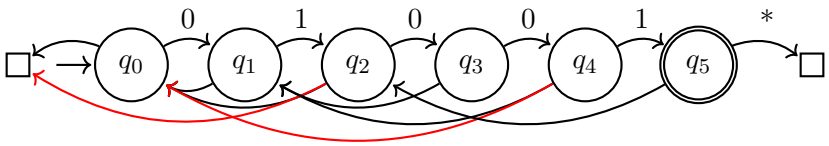

Figure 2: An example from Crochemore and Rytter (2002, §2.1). The MP tree for the pattern 01001 is drawn in black: right transitions are labelled with a symbol, whereas left transitions are unlabelled. The two 'K'-optimised left transitions are shown in red. The boxes denote *Null*. The root node is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

```
⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩⟨proof⟩
fixrec next :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree → ([:'a:] × [:'a:]) tree where
  next·[::]·t = t
| ⟦x ≠ ⊥; xs ≠ ⊥⟧ ⟹
  next·(x :# xs)·Null = Null
| ⟦x ≠ ⊥; xs ≠ ⊥⟧ ⟹
  next·(x :# xs)·(Node·(us, [::])·l·r) = Node·(us, [::])·l·r
| ⟦v ≠ ⊥; vs ≠ ⊥; x ≠ ⊥; xs ≠ ⊥⟧ ⟹
  next·(x :# xs)·(Node·(us, v :# vs)·l·r) = If eq·x·v then l else Node·(us, v :# vs)·l·r

fixrec — Bird p131 "an even simpler form", with the 'K' optimisation
    root2  :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
and op2    :: [:'a:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
and rep2   :: [:'a:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree
and left2  :: [:'a:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree
and right2 :: [:'a:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree
where
  [simp del]:
  root2·pat = rep2·pat·([::], pat)
| op2·pat·Null·x = root2·pat
| usvs ≠ ⊥ ⟹
  op2·pat·(Node·usvs·l·r)·x = If prefix·[:x:]·(csnd·usvs) then r else op2·pat·l·x
| [simp del]:
  rep2·pat·usvs = Node·usvs·(left2·pat·usvs)·(right2·pat·usvs)
| left2·pat·([::], vs) = next·vs·Null
| ⟦u ≠ ⊥; us ≠ ⊥⟧ ⟹
  left2·pat·(u :# us, vs) = next·vs·(sfoldl·(op2·pat)·(root2·pat)·us) — Note the use of op2 and next.
| right2·pat·(us, [::]) = Null — Unreachable
| ⟦v ≠ ⊥; vs ≠ ⊥⟧ ⟹
  right2·pat·(us, v :# vs) = rep2·pat·(us :@ [:v:], vs)

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

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

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

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

lemma *concrete-tree-KMP*:

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

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

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

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

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

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

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 $\to q_0 \xrightarrow{0} q_1 \xrightarrow{1} \overbrace{q_2 \to q_0 \to \Box}$. 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 $\mathtt{1}^n$ for instance.

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

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

We proceed with the formalities of the data refinement.

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

lemma *abs2-strict*[*simp*]:
abs2·⊥ = ⊥
abs2·*Null* = ⊥
⟨*proof*⟩

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

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

⟨*proof*⟩

lemma *next-Null*[*simp*]:
assumes $xs \neq \bot$
shows $next{\cdot}xs{\cdot}Null = Null$
⟨*proof*⟩

lemma *next-snil*[*simp*]:
assumes $xs \neq \bot$
shows $next{\cdot}xs{\cdot}(Node{\cdot}(us, [::]){\cdot}l{\cdot}r) = Node{\cdot}(us, [::]){\cdot}l{\cdot}r$
⟨*proof*⟩

lemma *op2-rep2-left2-right2-strict*[*simp*]:
$op2{\cdot}pat{\cdot}\bot = \bot$
$op2{\cdot}pat{\cdot}(Node{\cdot}(us, \bot){\cdot}l{\cdot}r) = \bot$
$op2{\cdot}pat{\cdot}(Node{\cdot}usvs{\cdot}l{\cdot}r){\cdot}\bot = \bot$
$rep2{\cdot}pat{\cdot}\bot = \bot$
$left2{\cdot}pat{\cdot}(\bot, vs) = \bot$
$left2{\cdot}pat{\cdot}(us, \bot) = \bot$
$right2{\cdot}pat{\cdot}(us, \bot) = \bot$
⟨*proof*⟩

lemma *snd-abs-root2-bottom*[*simp*]: $prod.snd\ (abs2{\cdot}(root2{\cdot}\bot)) = \bot$
⟨*proof*⟩

lemma *abs-rep2-ID'*[*simp*]: $abs2{\cdot}(rep2{\cdot}pat{\cdot}usvs) = usvs$
⟨*proof*⟩

lemma *abs-rep2-ID*: $abs2\ oo\ rep2{\cdot}pat = ID$
⟨*proof*⟩

lemma *rep2-snoc-right2*: — Bird p131
assumes $prefix{\cdot}[:x:]{\cdot}vs = TT$
shows $rep2{\cdot}pat{\cdot}(us\ :@\ [:x:], stail{\cdot}vs) = right2{\cdot}pat{\cdot}(us, vs)$
⟨*proof*⟩

lemma *not-prefix-op2-next*:
assumes $prefix{\cdot}[:x:]{\cdot}xs = FF$
shows $op2{\cdot}pat{\cdot}(next{\cdot}xs{\cdot}(rep2{\cdot}pat{\cdot}usvs)){\cdot}x = op2{\cdot}pat{\cdot}(rep2{\cdot}pat{\cdot}usvs){\cdot}x$
⟨*proof*⟩

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

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

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

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

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

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

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

lemma *op-op2-refinement*:
  assumes $pat \neq \bot$
  shows $rep2{\cdot}pat{\cdot}(op{\cdot}pat{\cdot}usvs{\cdot}x) = op2{\cdot}pat{\cdot}(rep2{\cdot}pat{\cdot}usvs){\cdot}x$
⟨*proof*⟩

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

lemma *data-refinement*:
  shows $matches = matches2$
⟨*proof*⟩

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

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

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

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

coinductive $K :: ([:'a::Eq:] \times [:'a:])\ tree \Rightarrow bool$ **where**
  $K\ Null$
| $[\![\ usvs \neq \bot;\ K\ l;\ K\ r;$
    $\bigwedge v\ vs.\ csnd{\cdot}usvs = v :\# vs \Longrightarrow l = Null \vee (\exists\, v'\ vs'.\ csnd{\cdot}(label{\cdot}l) = v' :\# vs' \wedge eq{\cdot}v{\cdot}v' = FF)$
  $]\!] \Longrightarrow K\ (Node{\cdot}usvs{\cdot}l{\cdot}r)$

declare *K.intros*[*intro*!, *simp*]

lemma *sfoldl-op2-root2-rep2-split*:
  assumes $pat \neq \bot$
  shows $sfoldl{\cdot}(op2{\cdot}pat){\cdot}(root2{\cdot}pat){\cdot}xs = rep2{\cdot}pat{\cdot}(split{\cdot}pat{\cdot}xs)$
⟨*proof*⟩

lemma *K-rep2*:
  assumes $pat \neq \bot$
  assumes $us :@ vs = pat$
  shows $K\ (rep2{\cdot}pat{\cdot}(us,\ vs))$
⟨*proof*⟩

theorem *K-root2*:
  assumes $pat \neq \bot$
  shows $K\ (root2{\cdot}pat)$
⟨*proof*⟩

The remaining steps are as follows:

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

- 8. discard *us*.
- 9. factor out *pat*.

4.3 Step 3: Introduce an accumulating parameter (grep)

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

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

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

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

lemma *r3-2*:
  (Λ (*root*, *op*, *rep*, *grep*). (*root*, *op*, *rep*))·
  (*root3* :: [:'a::*Eq-def*:] → ([:'a:] × [:'a:]) *tree*
  , *op3* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
  , *rep3* :: [:'a:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*
  , *grep3* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*)
 = (Λ (*root*, *op*, *rep*, *left*, *right*). (*root*, *op*, *rep*))·
  (*root2* :: [:'a::*Eq-def*:] → ([:'a:] × [:'a:]) *tree*
  , *op2* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
  , *rep2* :: [:'a::*Eq-def*:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*
  , *left2* :: [:'a::*Eq-def*:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*
  , *right2* :: [:'a::*Eq-def*:] → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*)
⟨*proof*⟩

4.4 Step 4: Inline rep

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

fixrec
  root4 :: [:'a::*Eq-def*:] → ([:'a:] × [:'a:]) *tree*
and *op4* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → 'a → ([:'a:] × [:'a:]) *tree*
and *grep4* :: [:'a:] → ([:'a:] × [:'a:]) *tree* → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) *tree*

where
  [*simp del*]:
  root4·*pat* = *grep4*·*pat*·*Null*·([::], *pat*)
| *op4*·*pat*·*Null*·*x* = *root4*·*pat*
| *usvs* ≠ ⊥ ⟹
   op4·*pat*·(*Node*·*usvs*·*l*·*r*)·*x* = *If prefix*·[:*x*:]·(*csnd*·*usvs*) *then r else op4*·*pat*·*l*·*x*
| [*simp del*]:
  grep4·*pat*·*l*·*usvs* = *Node*·*usvs*·(*next*·(*csnd*·*usvs*)·*l*)·(*case csnd*·*usvs of*
        [::] ⇒ *Null*
      | *v* :# *vs* ⇒ *grep4*·*pat*·(*case cfst*·*usvs* :@ [:*v*:] *of*
          [::] ⇒ *Null* — unreachable
        | *u* :# *us* ⇒ *sfoldl*·(*op4*·*pat*)·(*root4*·*pat*)·*us*)·(*cfst*·*usvs* :@ [:*v*:], *vs*))

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

lemma *fix-syn4-permute*:
  assumes *cont* (λ(*X1*, *X2*, *X3*, *X4*). *F1 X1 X2 X3 X4*)
  assumes *cont* (λ(*X1*, *X2*, *X3*, *X4*). *F2 X1 X2 X3 X4*)
  assumes *cont* (λ(*X1*, *X2*, *X3*, *X4*). *F3 X1 X2 X3 X4*)
  assumes *cont* (λ(*X1*, *X2*, *X3*, *X4*). *F4 X1 X2 X3 X4*)
  shows *fix-syn* (λ(*X1*, *X2*, *X3*, *X4*). (*F1 X1 X2 X3 X4*, *F2 X1 X2 X3 X4*, *F3 X1 X2 X3 X4*, *F4 X1 X2 X3 X4*))
    = (λ(*x1*, *x2*, *x4*, *x3*). (*x1*, *x2*, *x3*, *x4*))
      (*fix-syn* (λ(*X1*, *X2*, *X4*, *X3*). (*F1 X1 X2 X3 X4*, *F2 X1 X2 X3 X4*, *F4 X1 X2 X3 X4*, *F3 X1 X2 X3 X4*)))
⟨*proof*⟩

lemma *r4-3*:
  (*root4* :: [:'*a*::*Eq-def*:] → ([:'*a*:] × [:'*a*:]) *tree*
  , *op4* :: [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree* → '*a* → ([:'*a*:] × [:'*a*:]) *tree*
  , *grep4* :: [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree* → [:'*a*:] × [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree*)
 = (Λ (*root*, *op*, *rep*, *grep*). (*root*, *op*, *grep*))·
  (*root3* :: [:'*a*::*Eq-def*:] → ([:'*a*:] × [:'*a*:]) *tree*
  , *op3* :: [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree* → '*a* → ([:'*a*:] × [:'*a*:]) *tree*
  , *rep3* :: [:'*a*:] → [:'*a*:] × [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree*
  , *grep3* :: [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree* → [:'*a*:] × [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree*)
⟨*proof*⟩

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

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

fixrec
    root5 :: [:'*a*::*Eq-def*:] → ([:'*a*:] × [:'*a*:]) *tree*
and *op5* :: [:'*a*::*Eq-def*:] → ([:'*a*:] × [:'*a*:]) *tree* → '*a* → ([:'*a*:] × [:'*a*:]) *tree*
and *grep5* :: [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree* → [:'*a*:] × [:'*a*:] → ([:'*a*:] × [:'*a*:]) *tree*
where
  [*simp del*]:
  root5·*pat* = *grep5*·*pat*·*Null*·([::], *pat*)
| *op5*·*pat*·*Null*·*x* = *root5*·*pat*
| *usvs* ≠ ⊥ ⟹
   op5·*pat*·(*Node*·*usvs*·*l*·*r*)·*x* = *If prefix*·[:*x*:]·(*csnd*·*usvs*) *then r else op5*·*pat*·*l*·*x*
| [*simp del*]:

```
  grep5·pat·l·usvs = Node·usvs·(next·(csnd·usvs)·l)·(case csnd·usvs of
        [::] ⇒ Null
      | v :# vs ⇒ grep5·pat·(case cfst·usvs of — was cfst·usvs :@ [:v:]
          [::] ⇒ root5·pat
        | u :# us ⇒ sfoldl·(op5·pat)·(root5·pat)·(us :@ [:v:]))·(cfst·usvs :@ [:v:], vs))

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

lemma op5-grep5-strict[simp]:
  op5·pat·⊥ = ⊥
  op5·pat·(Node·(us, ⊥)·l·r) = ⊥
  op5·pat·(Node·usvs·l·r)·⊥ = ⊥
  grep5·pat·l·⊥ = ⊥
  ⟨proof⟩

lemma r5-4:
  ( root5  :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
  , op5    :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
  , grep5  :: [:'a:] → ([:'a:] × [:'a:]) tree → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree )
 = ( root4  :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree
  , op4    :: [:'a::Eq-def:] → ([:'a:] × [:'a:]) tree → 'a → ([:'a:] × [:'a:]) tree
  , grep4  :: [:'a:] → ([:'a:] × [:'a:]) tree → [:'a:] × [:'a:] → ([:'a:] × [:'a:]) tree )
⟨proof⟩
```

4.6 Step 6: Memoize left

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

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

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

lemma op6-grep6-strict[simp]:
  op6·pat·⊥ = ⊥
```

op6·pat·(*Node·*(*us*, ⊥)·*l·r*) = ⊥
op6·pat·(*Node·usvs·l·r*)·⊥ = ⊥
grep6·pat·l·⊥ = ⊥
⟨*proof*⟩

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

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

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

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

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

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

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

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

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

lemma *left6-left5-strict*[*simp*]:
left6·pat·⊥ = ⊥
left5·pat·⊥ = ⊥
⟨*proof*⟩

lemma *op6-left6*: ⟦*us* ≠ ⊥; *v* ≠ ⊥⟧ ⟹ *op6·pat·*(*left6·pat·us*)·*v* = *left6·pat·*(*us* :@ [:*v*:])
⟨*proof*⟩

lemma *op5-left5*: ⟦*us* ≠ ⊥; *v* ≠ ⊥⟧ ⟹ *op5·pat·*(*left5·pat·us*)·*v* = *left5·pat·*(*us* :@ [:*v*:])
⟨*proof*⟩

lemma *root5-left5*: *v* ≠ ⊥ ⟹ *root5·pat* = *left5·pat·*[:*v*:]
⟨*proof*⟩

lemma *op5-sfoldl-left5*: ⟦*us* ≠ ⊥; *u* ≠ ⊥; *v* ≠ ⊥⟧ ⟹

$op5{\cdot}pat{\cdot}(sfoldl{\cdot}(op5{\cdot}pat){\cdot}(root5{\cdot}pat){\cdot}us){\cdot}v = left5{\cdot}pat{\cdot}(u :\# us :@ [:v:])$
⟨*proof*⟩

lemma *root-bisim-root*:
  assumes $pat \neq \bot$
  shows *root-bisim pat* (*root6*·*pat*) (*root5*·*pat*)
⟨*proof*⟩

lemma *next-grep6-cases*[*consumes 3*, *case-names gl nl*]:
  assumes $vs \neq \bot$
  assumes $xs \neq \bot$
  assumes $P\ (next{\cdot}xs{\cdot}(grep6{\cdot}pat{\cdot}(left6{\cdot}pat{\cdot}us){\cdot}(us,\ vs)))$
  obtains (*gl*) $P\ (grep6{\cdot}pat{\cdot}(left6{\cdot}pat{\cdot}us){\cdot}(us,\ vs))$ | (*nl*) $P\ (next{\cdot}vs{\cdot}(left6{\cdot}pat{\cdot}us))$
⟨*proof*⟩

lemma *root-bisim-op-next56*:
  assumes *root-bisim pat t6 t5*
  assumes $prefix{\cdot}[:x:]{\cdot}xs = FF$
  shows $op6{\cdot}pat{\cdot}(next{\cdot}xs{\cdot}t6){\cdot}x = op6{\cdot}pat{\cdot}t6{\cdot}x \wedge op5{\cdot}pat{\cdot}(next{\cdot}xs{\cdot}t5){\cdot}x = op5{\cdot}pat{\cdot}t5{\cdot}x$
⟨*proof*⟩

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

lemma
  assumes $pat \neq \bot$
  shows *root-bisim-op*: *root-bisim pat t6 t5* $\Longrightarrow$ *root-bisim pat* ($op6{\cdot}pat{\cdot}t6{\cdot}x$) ($op5{\cdot}pat{\cdot}t5{\cdot}x$) — unused
    and *root-bisim-next-left*: *root-bisim pat* ($next{\cdot}vs{\cdot}(left6{\cdot}pat{\cdot}us)$) ($next{\cdot}vs{\cdot}(left5{\cdot}pat{\cdot}us)$) (**is** *?rbnl us vs*)
⟨*proof*⟩

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

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

lemma *tree-bisim-root-bisim*:
  shows *tree-bisim* (*root-bisim pat*)
⟨*proof*⟩

lemma *r6-5*:
  shows ($root6{\cdot}pat$, $op6{\cdot}pat$) = ($root5{\cdot}pat$, $op5{\cdot}pat$)
⟨*proof*⟩

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

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

4.7 Step 7: Simplify, unfold prefix

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

We also unfold *prefix* in *op6*.

fixrec
    $root7\ ::\ [:'a::Eq\text{-}def:] \to ([:'a:] \times [:'a:])\ tree$
and $op7\ \ ::\ [:'a::Eq\text{-}def:] \to ([:'a:] \times [:'a:])\ tree \to 'a \to ([:'a:] \times [:'a:])\ tree$

and *grep7* :: $[:'a:] \to ([:'a:] \times [:'a:])\ tree \to [:'a:] \times [:'a:] \to ([:'a:] \times [:'a:])\ tree$
where
[*simp del*]:
$root7{\cdot}pat = grep7{\cdot}pat{\cdot}Null{\cdot}([::],\ pat)$
| $op7{\cdot}pat{\cdot}Null{\cdot}x = root7{\cdot}pat$
| $op7{\cdot}pat{\cdot}(Node{\cdot}(us,\ [::]){\cdot}l{\cdot}r){\cdot}x =$ *op7*·*pat*·*l*·*x* — Unfold *prefix*
| $[\![v \neq \bot;\ vs \neq \bot]\!] \Longrightarrow$
$op7{\cdot}pat{\cdot}(Node{\cdot}(us,\ v :\# vs){\cdot}l{\cdot}r){\cdot}x = If\ eq{\cdot}x{\cdot}v\ then\ r\ else\ op7{\cdot}pat{\cdot}l{\cdot}x$
| [*simp del*]:
$grep7{\cdot}pat{\cdot}l{\cdot}(us,\ [::]) = Node{\cdot}(us,\ [::]){\cdot}l{\cdot}Null$ — Case split on *vs* hoisted above *Node*.
| $[\![v \neq \bot;\ vs \neq \bot]\!] \Longrightarrow$
$grep7{\cdot}pat{\cdot}l{\cdot}(us,\ v :\# vs) = Node{\cdot}(us,\ v :\# vs){\cdot}(next{\cdot}(v :\# vs){\cdot}l){\cdot}(grep7{\cdot}pat{\cdot}(op7{\cdot}pat{\cdot}l{\cdot}v){\cdot}(us :@ [:v:],\ vs))$

schematic-goal *root7-op7-grep7-def*:
(*root7* :: $[:'a::Eq\text{-}def:] \to ([:'a:] \times [:'a:])\ tree$
, *op7* :: $[:'a:] \to ([:'a:] \times [:'a:])\ tree \to 'a \to ([:'a:] \times [:'a:])\ tree$
, *grep7* :: $[:'a:] \to ([:'a:] \times [:'a:])\ tree \to [:'a:] \times [:'a:] \to ([:'a:] \times [:'a:])\ tree$)
= *fix*·*?F*
⟨*proof*⟩

lemma *r7-6-aux*:
assumes $pat \neq \bot$
shows
$(\Lambda\ (root,\ op,\ grep).\ (root{\cdot}pat,\ seq{\cdot}x{\cdot}(op{\cdot}pat{\cdot}t{\cdot}x),\ grep{\cdot}pat{\cdot}l{\cdot}(us,\ vs))){\cdot}$
(*root7* :: $[:'a::Eq\text{-}def:] \to ([:'a:] \times [:'a:])\ tree$
, *op7* :: $[:'a::Eq\text{-}def:] \to ([:'a:] \times [:'a:])\ tree \to 'a \to ([:'a:] \times [:'a:])\ tree$
, *grep7* :: $[:'a:] \to ([:'a:] \times [:'a:])\ tree \to [:'a:] \times [:'a:] \to ([:'a:] \times [:'a:])\ tree$)
$= (\Lambda\ (root,\ op,\ grep).\ (root{\cdot}pat,\ seq{\cdot}x{\cdot}(op{\cdot}pat{\cdot}t{\cdot}x),\ seq{\cdot}vs{\cdot}(grep{\cdot}pat{\cdot}l{\cdot}(us,\ vs)))){\cdot}$
(*root6* :: $[:'a::Eq\text{-}def:] \to ([:'a:] \times [:'a:])\ tree$
, *op6* :: $[:'a::Eq\text{-}def:] \to ([:'a:] \times [:'a:])\ tree \to 'a \to ([:'a:] \times [:'a:])\ tree$
, *grep6* :: $[:'a:] \to ([:'a:] \times [:'a:])\ tree \to [:'a:] \times [:'a:] \to ([:'a:] \times [:'a:])\ tree$)
⟨*proof*⟩

lemma *r7-6*:
assumes $pat \neq \bot$
shows $root7{\cdot}pat = root6{\cdot}pat$
and $x \neq \bot \Longrightarrow op7{\cdot}pat{\cdot}t{\cdot}x = op6{\cdot}pat{\cdot}t{\cdot}x$
⟨*proof*⟩

4.8 Step 8: Discard us

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

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

fixrec — Bird p133
root8 :: $[:'a::Eq\text{-}def:] \to [:'a:]\ tree$
and *op8* :: $[:'a:] \to [:'a:]\ tree \to 'a \to [:'a:]\ tree$
and *grep8* :: $[:'a:] \to [:'a:]\ tree \to [:'a:] \to [:'a:]\ tree$
where
[*simp del*]:
$root8{\cdot}pat = grep8{\cdot}pat{\cdot}Null{\cdot}pat$
| $op8{\cdot}pat{\cdot}Null{\cdot}x = root8{\cdot}pat$

| *op8·pat·(Node·[::]·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·[::] = Node·[::]·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*
⟨*proof*⟩

lemma *next′-strict*[*simp*]:
next′·x·⊥ = ⊥
next′·⊥·(Node·(v :# vs)·l·r) = ⊥
⟨*proof*⟩

lemma *root8-op8-grep8-strict*[*simp*]:
grep8·pat·l·⊥ = ⊥
op8·pat·⊥ = ⊥
root8·⊥ = ⊥
⟨*proof*⟩

lemma *ok8-strict*[*simp*]:
ok8·⊥ = ⊥
ok8·Null = ⊥
⟨*proof*⟩

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

lemma *next′-next*:
assumes *v ≠ ⊥*
assumes *vs ≠ ⊥*
shows *next′·v·(tree-map′·csnd·t) = tree-map′·csnd·(next·(v :# vs)·t)*
⟨*proof*⟩

lemma *r8-7*[*simplified*]:
shows (Λ (*root*, *op*, *grep*). (*root·pat*
, *op·pat·(tree-map′·csnd·t)·x*
, *grep·pat·(tree-map′·csnd·l)·(csnd·usvs*)))·(*root8*, *op8*, *grep8*)
= (Λ (*root*, *op*, *grep*). (*tree-map′·csnd·(root·pat)*
, *tree-map′·csnd·(op·pat·t·x)*
, *tree-map′·csnd·(grep·pat·l·usvs*)))·(*root7*, *op7*, *grep7*)
⟨*proof*⟩

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 $x \neq \bot$:

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

lemma *ok8-ok8*: *ok8 oo tree-map′·csnd = snull oo csnd oo abs2* (**is** *?lhs = ?rhs*)
⟨*proof*⟩

lemma *matches8*: — Bird p133
shows *matches·pat = smap·cfst oo sfilter·(ok8 oo csnd) oo sscanl·(Λ (n, x) y. (n + 1, op8·pat·x·y))·(0, root8·pat)*
(**is** *?lhs = ?rhs*)
⟨*proof*⟩

4.9 Step 9: Factor out pat (final version)

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

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

```
fixrec matchesf :: [:'a::Eq-def:] → [:'a:] → [:Integer:] where
[simp del]: matchesf·pat =
    (Letrec okf = (Λ (Node·vs·l·r). snull·vs);
            nextf = (Λ x t. case t of
                      Null ⇒ Null
                    | Node·vs·l·r ⇒ (case vs of
                        [::] ⇒ t
                      | v :# vs ⇒ If eq·x·v then l else t));
            rootf = grepf·Null·pat,
            opf = (Λ t x. case t of
                      Null ⇒ rootf
                    | Node·vs·l·r ⇒ (case vs of
                        [::] ⇒ opf·l·x
                      | v :# vs ⇒ If eq·x·v then r else opf·l·x)),
            grepf = (Λ l vs. case vs of
                      [::] ⇒ Node·[::]·l·Null
                    | v :# vs ⇒ Node·(v :# vs)·(nextf·v·l)·(grepf·(opf·l·v)·vs));
            stepf = (Λ (n, t) x. (n + 1, opf·t·x))
          in smap·cfst oo sfilter·(okf oo csnd) oo sscanl·stepf·(0, rootf))
```

lemma *matchesf-ok8*: (Λ (*Node·vs·l·r*). *snull·vs*) = *ok8*
⟨*proof*⟩

lemma *matchesf-next′*:
(Λ *x t. case t of Null ⇒ Null | Node·vs·l·r ⇒ (case vs of [::] ⇒ t | v :# vs ⇒ If eq·x·v then l else t)*) = *next′*
⟨*proof*⟩

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*)
⟨*proof*⟩

theorem *matches-final*:
shows *matches = matchesf*
⟨*proof*⟩

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 $g{\cdot}\bot = \bot$
  assumes $p{\cdot}\bot = \bot$
  shows $smap{\cdot}g\ oo\ sfilter{\cdot}p\ oo\ sscanl{\cdot}f{\cdot}z$
    $= (\mu\ R.\ \Lambda\ z\ xxs.\ case\ xxs\ of$
      $[::] \Rightarrow If\ p{\cdot}z\ then\ [:g{\cdot}z:]\ else\ [::]$
    $\mid x :\#\ xs \Rightarrow let\ z' = f{\cdot}z{\cdot}x\ in\ If\ p{\cdot}z\ then\ g{\cdot}z :\#\ R{\cdot}z'{\cdot}xs\ else\ R{\cdot}z'{\cdot}xs){\cdot}z$
(**is** *?lhs* = *?rhs*)
⟨*proof*⟩

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.

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

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

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

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

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

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

%% Driver

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

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

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

Figure 3: The final KMP program transliterated into Prolog.

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

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

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

type 'a lazy = 'a thunk ref

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

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

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

type 'a ltree = 'a tree lazy

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

end;
```

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

M. Tullsen. *PATH, a Program Transformation System for Haskell*. PhD thesis, Yale University, New Haven, CT, USA, 2002. URL http://www.cs.yale.edu/publications/techreports/tr1229.pdf.

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