Certification-Monads*

Christian Sternagel and René Thiemann

September 4, 2026

Abstract

This entry provides several monads intended for the development of stand-alone certifiers via code generation from Isabelle/HOL. More specifically, there are three flavors of error monads (the sum type, for the case where all monadic functions are total; an instance of the former, the so called check monad, yielding either success without any further information or an error message; as well as a variant of the sum type that accommodates partial functions by providing an explicit bottom element) and a parser monad built on top. All of this monads are heavily used in the IsaFoR/CeTA project which thus provides many examples of their usage.

Contents

1 Try-Catch and Error-Update Notation for Arbitrary Types 1

2 The Sum Type as Error Monad 2
2.1 Monad Laws 2
2.2 Monadic Map for Error Monad 7

3 A Special Error Monad for Certification with Informative Error Messages 8

4 A Sum Type with Bottom Element 12
4.1 Setup for Partial Functions 13
4.2 Monad Setup 13
4.3 Connection to *Partial-Function-MR.Partial-Function-MR* . . 16

5 Monadic Parser Combinators 16
5.1 Monad-Setup for Parsers 17

6 More material on parsing 24

*This research is supported by FWF (Austrian Science Fund) projects J3202 and P22767.

1 Try-Catch and Error-Update Notation for Arbitrary Types

theory *Error-Syntax*
imports
Main
begin

consts
catch :: $'a \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'c$ (‹(*try*(/ -)/ *catch*(/ -))› [*12*, *12*] *13*)
update-error :: $'a \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'd$ (**infixl** ‹<+?› *61*)

syntax
-replace-error :: $'a \Rightarrow 'b \Rightarrow 'a$ (**infixl** ‹<?› *61*)

syntax-consts
-replace-error ⇌ *update-error*

translations
m <? *e* ⇀ *m* <+? (λ-. *e*)

end

2 The Sum Type as Error Monad

theory *Error-Monad*
imports
HOL−Library.Monad-Syntax
Error-Syntax
begin

Make monad syntax (including do-notation) available for the sum type.

definition *bind* :: $'e + 'a \Rightarrow ('a \Rightarrow 'e + 'b) \Rightarrow 'e + 'b$
where
bind m f = (*case m of Inr x* ⇒ *f x* | *Inl e* ⇒ *Inl e*)

adhoc-overloading
Monad-Syntax.bind ⇌ *bind*

abbreviation (*input*) *return* ≡ *Inr*
abbreviation (*input*) *error* ≡ *Inl*
abbreviation (*input*) *run* ≡ *projr*

2.1 Monad Laws

lemma *return-bind* [*simp*]:
(*return x* ⤜ *f*) = *f x*
by (*simp add*: *bind-def*)

lemma *bind-return* [*simp*]:
(*m* ⤜ *return*) = *m*
by (*cases m*) (*simp-all add*: *bind-def*)

lemma *error-bind* [*simp*]:
(*error e* ⤜ *f*) = *error e*
by (*simp add*: *bind-def*)

lemma *bind-assoc* [*simp*]:
fixes *m* :: *'a* + *'b*
shows ((*m* ⤜ *f*) ⤜ *g*) = (*m* ⤜ (λ*x*. *f x* ⤜ *g*))
by (*cases m*) (*simp-all add*: *bind-def*)

lemma *bind-cong* [*fundef-cong*]:
fixes *m1 m2* :: *'e* + *'a*
and *f1 f2* :: *'a* ⇒ *'e* + *'b*
assumes *m1* = *m2*
and ⋀*y*. *m2* = *Inr y* ⟹ *f1 y* = *f2 y*
shows (*m1* ⤜ *f1*) = (*m2* ⤜ *f2*)
using *assms* **by** (*cases m1*) (*auto simp*: *bind-def*)

definition *catch-error* :: *'e* + *'a* ⇒ (*'e* ⇒ *'f* + *'a*) ⇒ *'f* + *'a*
where
catch-def: *catch-error m f* = (*case m of Inl e* ⇒ *f e* | *Inr x* ⇒ *Inr x*)

adhoc-overloading
Error-Syntax.catch ⇌ *catch-error*

lemma *catch-splits*:
P (*try m catch f*) ⟷ (∀ *e*. *m* = *Inl e* ⟶ *P* (*f e*)) ∧ (∀ *x*. *m* = *Inr x* ⟶ *P* (*Inr x*))
P (*try m catch f*) ⟷ (¬ ((∃ *e*. *m* = *Inl e* ∧ ¬ *P* (*f e*)) ∨ (∃ *x*. *m* = *Inr x* ∧ ¬ *P* (*Inr x*))))
by (*case-tac* [!] *m*) (*simp-all add*: *catch-def*)

abbreviation *update-error* :: *'e* + *'a* ⇒ (*'e* ⇒ *'f*) ⇒ *'f* + *'a*
where
update-error m f ≡ *try m catch* (λ*x*. *error* (*f x*))

adhoc-overloading
Error-Syntax.update-error ⇌ *update-error*

lemma *catch-return* [*simp*]:
(*try return x catch f*) = *return x* **by** (*simp add*: *catch-def*)

lemma *catch-error* [*simp*]:
(*try error e catch f*) = *f e* **by** (*simp add*: *catch-def*)

lemma *update-error-return* [*simp*]:
(*m* <+? *c* = *return x*) ⟷ (*m* = *return x*)
by (*cases m*) *simp-all*

definition *isOK m* ⟷ (*case m of Inl e* ⇒ *False* | *Inr x* ⇒ *True*)

lemma *isOK-E* [*elim*]:
assumes *isOK m*
obtains *x* **where** *m* = *return x*
using *assms* **by** (*cases m*) (*simp-all add*: *isOK-def*)

lemma *isOK-I* [*simp*, *intro*]:
m = *return x* ⟹ *isOK m*
by (*cases m*) (*simp-all add*: *isOK-def*)

lemma *isOK-iff*:
isOK m ⟷ (∃ *x*. *m* = *return x*)
by *blast*

lemma *isOK-error* [*simp*]:
isOK (*error x*) = *False*
by *blast*

lemma *isOK-bind* [*simp*]:
isOK (*m* ⤜ *f*) ⟷ *isOK m* ∧ *isOK* (*f* (*run m*))
by (*cases m*) *simp-all*

lemma *isOK-update-error* [*simp*]:
isOK (*m* <+? *f*) ⟷ *isOK m*
by (*cases m*) *simp-all*

lemma *isOK-case-prod* [*simp*]:
isOK (*case lr of* (*l*, *r*) ⇒ *P l r*) = (*case lr of* (*l*, *r*) ⇒ *isOK* (*P l r*))
by (*rule prod.case-distrib*)

lemma *isOK-case-option* [*simp*]:
isOK (*case x of None* ⇒ *P* | *Some v* ⇒ *Q v*) = (*case x of None* ⇒ *isOK P* |
Some v ⇒ *isOK* (*Q v*))
by (*cases x*) (*auto*)

lemma *isOK-Let* [*simp*]:
isOK (*Let s f*) = *isOK* (*f s*)
by (*simp add*: *Let-def*)

lemma *run-bind* [*simp*]:
isOK m ⟹ *run* (*m* ⤜ *f*) = *run* (*f* (*run m*))
by *auto*

lemma *run-catch* [*simp*]:

```
  isOK m ⟹ run (try m catch f) = run m
  by auto

fun foldM :: ('a ⇒ 'b ⇒ 'e + 'a) ⇒ 'a ⇒ 'b list ⇒ 'e + 'a
where
  foldM f d [] = return d |
  foldM f d (x # xs) = do { y ← f d x; foldM f y xs }

fun forallM-index-aux :: ('a ⇒ nat ⇒ 'e + unit) ⇒ nat ⇒ 'a list ⇒ (('a × nat)
× 'e) + unit
where
  forallM-index-aux P i [] = return () |
  forallM-index-aux P i (x # xs) = do {
    P x i <+? Pair (x, i);
    forallM-index-aux P (Suc i) xs
  }

lemma isOK-forallM-index-aux [simp]:
  isOK (forallM-index-aux P n xs) = (∀ i < length xs. isOK (P (xs ! i) (i + n)))
proof (induct xs arbitrary: n)
  case (Cons x xs)
  have (∀ i < length (x # xs). isOK (P ((x # xs) ! i) (i + n))) ⟷
    (isOK (P x n) ∧ (∀ i < length xs. isOK (P (xs ! i) (i + Suc n))))
    by (auto, case-tac i) (simp-all)
  then show ?case
    unfolding Cons [of Suc n, symmetric] by simp
qed auto

definition forallM-index :: ('a ⇒ nat ⇒ 'e + unit) ⇒ 'a list ⇒ (('a × nat) × 'e)
+ unit
where
  forallM-index P xs = forallM-index-aux P 0 xs

lemma isOK-forallM-index [simp]:
  isOK (forallM-index P xs) ⟷ (∀ i < length xs. isOK (P (xs ! i) i))
  unfolding forallM-index-def isOK-forallM-index-aux by simp

lemma forallM-index [fundef-cong]:
  fixes c :: 'a ⇒ nat ⇒ 'e + unit
  assumes ⋀x i. x ∈ set xs ⟹ c x i = d x i
  shows forallM-index c xs = forallM-index d xs
proof −
  { fix n
    have forallM-index-aux c n xs = forallM-index-aux d n xs
      using assms by (induct xs arbitrary: n) simp-all }
  then show ?thesis by (simp add: forallM-index-def)
qed

hide-const forallM-index-aux
```

Check whether f succeeds for all elements of a given list. In case it doesn't, return the first offending element together with the produced error.

fun *forallM* :: $('a \Rightarrow 'e + unit) \Rightarrow 'a\ list \Rightarrow ('a * 'e) + unit$
where
  $forallM\ f\ [] = return\ ()$ |
  $forallM\ f\ (x\ \#\ xs) = f\ x <+?\ Pair\ x \gg forallM\ f\ xs$

lemma *forallM-fundef-cong* [*fundef-cong*]:
  assumes $xs = ys\ \bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
  shows $forallM\ f\ xs = forallM\ g\ ys$
  unfolding *assms*(*1*) **using** *assms*(*2*)
proof (*induct ys*)
  case (*Cons x xs*)
  thus *?case* **by** (*cases g x*, *auto*)
qed *auto*

lemma *isOK-forallM* [*simp*]:
  $isOK\ (forallM\ f\ xs) \longleftrightarrow (\forall x \in set\ xs.\ isOK\ (f\ x))$
  by (*induct xs*) (*simp-all*)

Check whether f succeeds for at least one element of a given list. In case it doesn't, return the list of produced errors.

fun *existsM* :: $('a \Rightarrow 'e + unit) \Rightarrow 'a\ list \Rightarrow 'e\ list + unit$
where
  $existsM\ f\ [] = error\ []$ |
  $existsM\ f\ (x\ \#\ xs) = (try\ f\ x\ catch\ (\lambda e.\ existsM\ f\ xs <+?\ Cons\ e))$

lemma *existsM-cong* [*fundef-cong*]:
  assumes $xs = ys$
  and $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
  shows $existsM\ f\ xs = existsM\ g\ ys$
  using *assms*
  by (*induct ys arbitrary*:*xs*) (*auto split*:*catch-splits*)

lemma *isOK-existsM* [*simp*]:
  $isOK\ (existsM\ f\ xs) \longleftrightarrow (\exists x \in set\ xs.\ isOK\ (f\ x))$
proof (*induct xs*)
  case (*Cons x xs*)
  show *?case*
  proof (*cases f x*)
    case (*Inl e*)
    with *Cons* **show** *?thesis* **by** *simp*
  qed (*auto simp add*: *catch-def*)
qed *simp*

lemma *is-OK-if-return* [*simp*]:
  $isOK\ (if\ b\ then\ return\ x\ else\ m) \longleftrightarrow b \vee isOK\ m$
  $isOK\ (if\ b\ then\ m\ else\ return\ x) \longleftrightarrow \neg\ b \vee isOK\ m$
  by *simp-all*

lemma *isOK-if-error* [*simp*]:
isOK (*if b then error e else m*) ⟷ ¬ *b* ∧ *isOK m*
isOK (*if b then m else error e*) ⟷ *b* ∧ *isOK m*
by *simp-all*

lemma *isOK-if*:
isOK (*if b then x else y*) ⟷ *b* ∧ *isOK x* ∨ ¬ *b* ∧ *isOK y*
by *simp*

fun *sequence* :: (*'e* + *'a*) *list* ⇒ *'e* + *'a list*
where
sequence [] = *Inr* [] |
sequence (*m* # *ms*) = *do* {
x ← *m*;
xs ← *sequence ms*;
return (*x* # *xs*)
}

2.2 Monadic Map for Error Monad

fun *mapM* :: (*'a* ⇒ *'e* + *'b*) ⇒ *'a list* ⇒ *'e* + *'b list*
where
mapM f [] = *return* [] |
mapM f (*x*#*xs*) = *do* {
y ← *f x*;
ys ← *mapM f xs*;
Inr (*y* # *ys*)
}

lemma *mapM-error*:
(∃ *e*. *mapM f xs* = *error e*) ⟷ (∃ *x*∈*set xs*. ∃ *e*. *f x* = *error e*)
proof (*induct xs*)
case (*Cons x xs*)
then show *?case*
by (*cases f x*, *simp-all*, *cases mapM f xs*, *simp-all*)
qed *simp*

lemma *mapM-return*:
assumes *mapM f xs* = *return ys*
shows *ys* = *map* (*run* ∘ *f*) *xs* ∧ (∀ *x*∈*set xs*. ∀ *e*. *f x* ≠ *error e*)
using *assms*
proof (*induct xs arbitrary*: *ys*)
case (*Cons x xs ys*)
then show *?case*
by (*cases f x*, *simp*, *cases mapM f xs*, *simp-all*)
qed *simp*

lemma *mapM-return-idx*:

assumes $*$: *mapM f xs* = *Inr ys* **and** *i* < *length xs*
shows $\exists\, y.\ f\ (xs\ !\ i) = Inr\ y \wedge ys\ !\ i = y$
proof −
note $** = mapM\text{-}return\ [OF\ *,\ unfolded\ set\text{-}conv\text{-}nth]$
with *assms* **have** $\bigwedge e.\ f\ (xs\ !\ i) \neq Inl\ e$ **by** *auto*
then obtain *y* **where** *f* (*xs* ! *i*) = *Inr y* **by** (*cases f* (*xs* ! *i*)) *auto*
then have *f* (*xs* ! *i*) = *Inr y* ∧ *ys* ! *i* = *y* **unfolding** $** [THEN\ conjunct1]$
using *assms* **by** *auto*
then show *?thesis* **..**
qed

lemma *mapM-cong* [*fundef-cong*]:
assumes *xs* = *ys* **and** $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
shows *mapM f xs* = *mapM g ys*
unfolding *assms*(*1*) **using** *assms*(*2*) **by** (*induct ys*) *auto*

lemma *bindE* [*elim*]:
assumes (*p* ⤜ *f*) = *return x*
obtains *y* **where** *p* = *return y* **and** *f y* = *return x*
using *assms* **by** (*cases p*) *simp-all*

lemma *then-return-eq* [*simp*]:
(*p* ≫ *q*) = *return f* ⟷ *isOK p* ∧ *q* = *return f*
by (*cases p*) *simp-all*

fun *choice* :: ('*e* + '*a*) *list* ⇒ '*e list* + '*a*
where
choice [] = *error* [] |
choice (*x* # *xs*) = (*try x catch* (λ*e*. *choice xs* <+*? Cons e*))

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

lemma *isOK-mapM*:
assumes *isOK* (*mapM f xs*)
shows (∀ *x*. *x* ∈ *set xs* ⟶ *isOK* (*f x*)) ∧ *run* (*mapM f xs*) = *map* (λ*x*. *run* (*f x*)) *xs*
using *assms mapM-return*[*of f xs*] **by** (*force simp*: *isOK-def split*: *sum.splits*)+

fun *firstM*
where
firstM f [] = *error* []
| *firstM f* (*x* # *xs*) = (*try f x* ≫ *return x catch* (λ*e*. *firstM f xs* <+*? Cons e*))

lemma *firstM*:
isOK (*firstM f xs*) ⟷ (∃ *x*∈*set xs*. *isOK* (*f x*))
by (*induct xs*) (*auto simp*: *catch-def split*: *sum.splits*)

lemma *firstM-return*:
assumes *firstM f xs* = *return y*

shows *isOK* (*f y*) ∧ *y* ∈ *set xs*
using *assms* **by** (*induct xs*) (*auto simp*: *catch-def split*: *sum.splits*)

end

3 A Special Error Monad for Certification with Informative Error Messages

theory *Check-Monad*
imports *Error-Monad*
begin

A check is either successful or fails with some error.

type-synonym
'e check = *'e* + *unit*

abbreviation *succeed* :: *'e check*
where
succeed ≡ *return* ()

definition *check* :: *bool* ⇒ *'e* ⇒ *'e check*
where
check b e = (*if b then succeed else error e*)

lemma *isOK-check* [*simp*]:
isOK (*check b e*) = *b* **by** (*simp add*: *check-def*)

lemma *isOK-check-catch* [*simp*]:
isOK (*try check b e catch f*) ⟷ *b* ∨ *isOK* (*f e*)
by (*auto simp add*: *catch-def check-def*)

definition *check-return* :: *'a check* ⇒ *'b* ⇒ *'a* + *'b*
where
check-return chk res = (*chk* ≫ *return res*)

lemma *check-return* [*simp*]:
check-return chk res = *return res'* ⟷ *isOK chk* ∧ *res'* = *res*
unfolding *check-return-def* **by** (*cases chk*) *auto*

lemma [*code-unfold*]:
check-return chk res = (*case chk of Inr* - ⇒ *Inr res* | *Inl e* ⇒ *Inl e*)
unfolding *check-return-def bind-def* **..**

abbreviation *check-allm* :: (*'a* ⇒ *'e check*) ⇒ *'a list* ⇒ *'e check*
where
check-allm f xs ≡ *forallM f xs* <+? *snd*

abbreviation *check-exm* :: ('*a* ⇒ '*e check*) ⇒ '*a list* ⇒ ('*e list* ⇒ '*e*) ⇒ '*e check*
where
check-exm f xs fld ≡ *existsM f xs* <+? *fld*

lemma *isOK-check-allm*:
isOK (*check-allm f xs*) ⟷ (∀ *x* ∈ *set xs*. *isOK* (*f x*))
by *simp*

abbreviation *check-allm-index* :: ('*a* ⇒ *nat* ⇒ '*e check*) ⇒ '*a list* ⇒ '*e check*
where
check-allm-index f xs ≡ *forallM-index f xs* <+? *snd*

abbreviation *check-all* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a check*
where
check-all f xs ≡ *check-allm* (λ*x*. *if f x then succeed else error x*) *xs*

abbreviation *check-all-index* :: ('*a* ⇒ *nat* ⇒ *bool*) ⇒ '*a list* ⇒ ('*a* × *nat*) *check*
where
check-all-index f xs ≡ *check-allm-index* (λ*x i*. *if f x i then succeed else error* (*x*, *i*)) *xs*

lemma *isOK-check-all-index* [*simp*]:
isOK (*check-all-index f xs*) ⟷ (∀ *i* < *length xs*. *f* (*xs* ! *i*) *i*)
by *auto*

The following version allows to modify the index during the check.

definition
check-allm-gen-index ::
('*a* ⇒ *nat* ⇒ *nat*) ⇒ ('*a* ⇒ *nat* ⇒ '*e check*) ⇒ *nat* ⇒ '*a list* ⇒ '*e check*
where
check-allm-gen-index g f n xs = *snd* (*foldl* (λ(*i*, *m*) *x*. (*g x i*, *m* ≫ *f x i*)) (*n*, *succeed*) *xs*)

lemma *foldl-error*:
snd (*foldl* (λ(*i*, *m*) *x* . (*g x i*, *m* ≫ *f x i*)) (*n*, *error e*) *xs*) = *error e*
by (*induct xs arbitrary*: *n*) *auto*

lemma *isOK-check-allm-gen-index* [*simp*]:
assumes *isOK* (*check-allm-gen-index g f n xs*)
shows ∀ *x*∈*set xs*. ∃ *i*. *isOK* (*f x i*)
using *assms*
proof (*induct xs arbitrary*: *n*)
case (*Cons x xs*)
show *?case*
proof (*cases isOK* (*f x n*))
case *True*
then have ∃ *i*. *isOK* (*f x i*) **by** *auto*
with *True Cons* **show** *?thesis*
unfolding *check-allm-gen-index-def* **by** (*force simp*: *isOK-iff*)

```
  next
    case False
    then obtain e where f x n = error e by (cases f x n) auto
    with foldl-error [of g f - e] and Cons show ?thesis
      unfolding check-allm-gen-index-def by auto
  qed
qed simp

lemma check-allm-gen-index [fundef-cong]:
  fixes f :: 'a ⇒ nat ⇒ 'e check
  assumes ⋀x n. x ∈ set xs ⟹ g x n = g' x n
    and ⋀x n. x ∈ set xs ⟹ f x n = f' x n
  shows check-allm-gen-index g f n xs = check-allm-gen-index g' f' n xs
proof −
  { fix n m
    have foldl (λ(i, m) x. (g x i, m ≫ f x i)) (n, m) xs =
      foldl (λ(i, m) x. (g' x i, m ≫ f' x i)) (n, m) xs
      using assms by (induct xs arbitrary: n m) auto }
  then show ?thesis unfolding check-allm-gen-index-def by simp
qed

definition check-subseteq :: 'a list ⇒ 'a list ⇒ 'a check
where
  check-subseteq xs ys = check-all (λx. x ∈ set ys) xs

lemma isOK-check-subseteq [simp]:
  isOK (check-subseteq xs ys) ⟷ set xs ⊆ set ys
  by (auto simp: check-subseteq-def)

definition check-same-set :: 'a list ⇒ 'a list ⇒ 'a check
where
  check-same-set xs ys = (check-subseteq xs ys ≫ check-subseteq ys xs)

lemma isOK-check-same-set [simp]:
  isOK (check-same-set xs ys) ⟷ set xs = set ys
  unfolding check-same-set-def by auto

definition check-disjoint :: 'a list ⇒ 'a list ⇒ 'a check
where
  check-disjoint xs ys = check-all (λx. x ∉ set ys) xs

lemma isOK-check-disjoint [simp]:
  isOK (check-disjoint xs ys) ⟷ set xs ∩ set ys = {}
  unfolding check-disjoint-def by (auto)

definition check-all-combinations :: ('a ⇒ 'a ⇒ 'b check) ⇒ 'a list ⇒ 'b check
where
  check-all-combinations c xs = check-allm (λx. check-allm (c x) xs) xs
```

lemma *isOK-check-all-combinations* [*simp*]:
isOK (*check-all-combinations c xs*) ⟷ (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set xs*. *isOK* (*c x y*))
unfolding *check-all-combinations-def* **by** *simp*

fun *check-pairwise* :: ('*a* ⇒ '*a* ⇒ '*b check*) ⇒ '*a list* ⇒ '*b check*
where
check-pairwise c [] = *succeed* |
check-pairwise c (*x* # *xs*) = (*check-allm* (*c x*) *xs* ≫ *check-pairwise c xs*)

lemma *pairwise-aux*:
(∀ *j*<*length* (*x* # *xs*). ∀ *i*<*j*. *P* ((*x* # *xs*) ! *i*) ((*x* # *xs*) ! *j*))
= ((∀ *j*<*length xs*. *P x* (*xs* ! *j*)) ∧ (∀ *j*<*length xs*. ∀ *i*<*j*. *P* (*xs* ! *i*) (*xs* ! *j*)))
(**is** *?C* = (*?A* ∧ *?B*))
proof (*intro iffI conjI*)
assume ∗: *?A* ∧ *?B*
show *?C*
proof (*intro allI impI*)
fix *i j*
assume *j* < *length* (*x* # *xs*) **and** *i* < *j*
then show *P* ((*x* # *xs*) ! *i*) ((*x* # *xs*) ! *j*)
proof (*induct j*)
case (*Suc j*)
then show *?case*
using ∗ **by** (*induct i*) *simp-all*
qed *simp*
qed
qed *force+*

lemma *isOK-check-pairwise* [*simp*]:
isOK (*check-pairwise c xs*) ⟷ (∀ *j*<*length xs*. ∀ *i*<*j*. *isOK* (*c* (*xs* ! *i*) (*xs* ! *j*)))
proof (*induct xs*)
case (*Cons x xs*)
have *isOK* (*check-allm* (*c x*) *xs*) = (∀ *j*<*length xs*. *isOK* (*c x* (*xs* ! *j*)))
using *all-set-conv-all-nth* [*of xs* λ*y*. *isOK* (*c x y*)] **by** *simp*
then have *isOK* (*check-pairwise c* (*x* # *xs*)) =
((∀ *j*<*length xs*. *isOK* (*c x* (*xs* ! *j*))) ∧ (∀ *j*<*length xs*. ∀ *i*<*j*. *isOK* (*c* (*xs* ! *i*)
(*xs* ! *j*))))
by (*simp add*: *Cons*)
then show *?case* **using** *pairwise-aux* [*of x xs* λ*x y*. *isOK* (*c x y*)] **by** *simp*
qed *auto*

abbreviation *check-exists* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ ('*a list*) *check*
where
check-exists f xs ≡ *check-exm* (λ*x*. *if f x then succeed else error* [*x*]) *xs concat*

lemma *isOK-choice* [*simp*]:
isOK (*choice* []) ⟷ *False*
isOK (*choice* (*x* # *xs*)) ⟷ *isOK x* ∨ *isOK* (*choice xs*)
by (*auto simp*: *choice.simps isOK-def split*: *sum.splits*)

fun *or-ok* :: *'a check* ⇒ *'a check* ⇒ *'a check* **where**
  or-ok (*Inl a*) *b* = *b* |
  or-ok (*Inr a*) *b* = *Inr a*

lemma *or-is-or*[*simp*]: *isOK* (*or-ok a b*) ⟷ *isOK a* ∨ *isOK b* **by** (*cases a*, *auto*)

end

4 A Sum Type with Bottom Element

theory *Strict-Sum*
imports
  HOL−Library.Monad-Syntax
  Error-Syntax
  Partial-Function-MR.Partial-Function-MR
begin

datatype (*dead 'e*, *'a*) *sum-bot* (**infixr** ‹+⊥› *10*) = *Bottom* | *Left 'e* | *Right 'a*
for *map*: *sum-bot-map*

4.1 Setup for Partial Functions

abbreviation *sum-bot-ord* :: *'e* +⊥ *'a* ⇒ *'e* +⊥ *'a* ⇒ *bool*
where
  sum-bot-ord ≡ *flat-ord Bottom*

interpretation *sum-bot*:
  partial-function-definitions sum-bot-ord flat-lub Bottom
  by (*rule flat-interpretation*)

declaration ‹
Partial-Function.init
  sum-bot
  @{*term sum-bot.fixp-fun*}
  @{*term sum-bot.mono-body*}
  @{*thm sum-bot.fixp-rule-uc*}
  @{*thm sum-bot.fixp-induct-uc*}
  NONE
›

4.2 Monad Setup

fun *bind* :: *'e* +⊥ *'a* ⇒ (*'a* ⇒ (*'e* +⊥ *'b*)) ⇒ *'e* +⊥ *'b*
where
  bind Bottom f = *Bottom* |
  bind (*Left e*) *f* = *Left e* |
  bind (*Right x*) *f* = *f x*

lemma *bind-cong* [*fundef-cong*]:
assumes $xs = ys$ **and** $\bigwedge x.\ ys = Right\ x \Longrightarrow f\ x = g\ x$
shows $bind\ xs\ f = bind\ ys\ g$
using *assms* **by** (*cases ys*) *simp-all*

abbreviation *mono-sum-bot* :: $(('a \Rightarrow ('e +_\bot 'b)) \Rightarrow 'f +_\bot 'c) \Rightarrow bool$
where
$mono\text{-}sum\text{-}bot \equiv monotone\ (fun\text{-}ord\ sum\text{-}bot\text{-}ord)\ sum\text{-}bot\text{-}ord$

lemma *bind-mono* [*partial-function-mono*]:
assumes *mf*: *mono-sum-bot B* **and** *mg*: $\bigwedge y.\ mono\text{-}sum\text{-}bot\ (\lambda f.\ C\ y\ f)$
shows $mono\text{-}sum\text{-}bot\ (\lambda f.\ bind\ (B\ f)\ (\lambda y.\ C\ y\ f))$
proof (*rule monotoneI*)
fix $f\ g :: 'a \Rightarrow 'b +_\bot 'c$
assume *fg*: *fun-ord sum-bot-ord f g*
with *mf* **have** *sum-bot-ord* $(B\ f)\ (B\ g)$ **by** (*rule monotoneD* [*of - - - f g*])
then have *sum-bot-ord* $(bind\ (B\ f)\ (\lambda y.\ C\ y\ f))\ (bind\ (B\ g)\ (\lambda y.\ C\ y\ f))$
unfolding *flat-ord-def* **by** *auto*
also from *mg* **have** $\bigwedge y'.\ sum\text{-}bot\text{-}ord\ (C\ y'\ f)\ (C\ y'\ g)$
by (*rule monotoneD*) (*rule fg*)
then have *sum-bot-ord* $(bind\ (B\ g)\ (\lambda y'.\ C\ y'\ f))\ (bind\ (B\ g)\ (\lambda y'.\ C\ y'\ g))$
unfolding *flat-ord-def* **by** (*cases B g*) *auto*
finally (*sum-bot.leq-trans*)
show *sum-bot-ord* $(bind\ (B\ f)\ (\lambda y.\ C\ y\ f))\ (bind\ (B\ g)\ (\lambda y'.\ C\ y'\ g))$ **.**
qed

adhoc-overloading
Monad-Syntax.bind $\rightleftharpoons$ *bind*

hide-const (**open**) *bind*

fun *catch-error* :: $'e +_\bot 'a \Rightarrow ('e \Rightarrow ('f +_\bot 'a)) \Rightarrow 'f +_\bot 'a$
where
catch-error Bottom f = *Bottom* |
catch-error (*Left a*) *f* = *f a* |
catch-error (*Right a*) *f* = *Right a*

adhoc-overloading
Error-Syntax.catch $\rightleftharpoons$ *catch-error*

lemma *catch-mono* [*partial-function-mono*]:
assumes *mf*: *mono-sum-bot B* **and** *mg*: $\bigwedge y.\ mono\text{-}sum\text{-}bot\ (\lambda f.\ C\ y\ f)$
shows $mono\text{-}sum\text{-}bot\ (\lambda f.\ try\ (B\ f)\ catch\ (\lambda y.\ C\ y\ f))$
proof (*rule monotoneI*)
fix $f\ g :: 'a \Rightarrow 'b +_\bot 'c$
assume *fg*: *fun-ord sum-bot-ord f g*
with *mf* **have** *sum-bot-ord* $(B\ f)\ (B\ g)$ **by** (*rule monotoneD* [*of - - - f g*])
then have *sum-bot-ord* $(try\ (B\ f)\ catch\ (\lambda y.\ C\ y\ f))\ (try\ (B\ g)\ catch\ (\lambda y.\ C\ y$

```
f))
    unfolding flat-ord-def by auto
  also from mg
  have ⋀y'. sum-bot-ord (C y' f) (C y' g)
    by (rule monotoneD) (rule fg)
  then have sum-bot-ord (try (B g) catch (λy'. C y' f)) (try (B g) catch (λy'. C
y' g))
    unfolding flat-ord-def by (cases B g) auto
  finally (sum-bot.leq-trans)
    show sum-bot-ord (try (B f) catch (λy. C y f)) (try (B g) catch (λy'. C y' g))
.
qed

definition error :: 'e ⇒ 'e +⊥ 'a
where
  [simp]: error x = Left x

definition return :: 'a ⇒ 'e +⊥ 'a
where
  [simp]: return x = Right x

fun map-sum-bot :: ('a ⇒ ('e +⊥ 'b)) ⇒ 'a list ⇒ 'e +⊥ 'b list
where
  map-sum-bot f [] = return [] |
  map-sum-bot f (x#xs) = do {
    y ← f x;
    ys ← map-sum-bot f xs;
    return (y # ys)
  }

lemma map-sum-bot-cong [fundef-cong]:
  assumes xs = ys and ⋀x. x ∈ set ys ⟹ f x = g x
  shows map-sum-bot f xs = map-sum-bot g ys
  unfolding assms(1) using assms(2) by (induct ys) auto

lemmas sum-bot-const-mono =
  sum-bot.const-mono [of fun-ord sum-bot-ord]

lemma map-sum-bot-mono [partial-function-mono]:
  fixes C :: 'a ⇒ ('b ⇒ ('e +⊥ 'c)) ⇒ 'e +⊥ 'd
  assumes ⋀y. y ∈ set B ⟹ mono-sum-bot (C y)
  shows mono-sum-bot (λf. map-sum-bot (λy. C y f) B)
  using assms by (induct B) (auto intro!: partial-function-mono)

abbreviation update-error :: 'e +⊥ 'a ⇒ ('e ⇒ 'f) ⇒ 'f +⊥ 'a
where
  update-error r f ≡ try r catch (λ e. error (f e))

adhoc-overloading
```

Error-Syntax.update-error ⇌ *update-error*

fun *sumbot* :: *'e* + *'a* ⇒ *'e* +⊥ *'a*
where
sumbot (*Inl x*) = *Left x* |
sumbot (*Inr x*) = *Right x*

code-datatype *sumbot*

lemma [*code*]:
bind (*sumbot a*) *f* = (*case a of Inl b* ⇒ *sumbot* (*Inl b*) | *Inr a* ⇒ *f a*)
by (*cases a*) *auto*

lemma [*code*]:
(*try* (*sumbot a*) *catch f*) = (*case a of Inl b* ⇒ *f b* | *Inr a* ⇒ *sumbot* (*Inr a*))
by (*cases a*) *auto*

lemma [*code*]: *Right x* = *sumbot* (*Inr x*) **by** *simp*

lemma [*code*]: *Left x* = *sumbot* (*Inl x*) **by** *simp*

lemma [*code*]: *return x* = *sumbot* (*Inr x*) **by** *simp*

lemma [*code*]: *error x* = *sumbot* (*Inl x*) **by** *simp*

lemma [*code*]:
case-sum-bot f g h (*sumbot p*) = *case-sum g h p*
by (*cases p*) *auto*

4.3 Connection to *Partial-Function-MR.Partial-Function-MR*

lemma *sum-bot-map-mono* [*partial-function-mono*]:
assumes *mf*: *mono-sum-bot B*
shows *mono-sum-bot* (λ*f*. *sum-bot-map h* (*B f*))
proof (*rule monotoneI*)
fix *f g* :: *'a* ⇒ *'b* +⊥ *'c*
assume *fg*: *fun-ord sum-bot-ord f g*
with *mf* **have** *sum-bot-ord* (*B f*) (*B g*) **by** (*rule monotoneD* [*of - - - f g*])
then show *sum-bot-ord* (*sum-bot-map h* (*B f*)) (*sum-bot-map h* (*B g*))
unfolding *flat-ord-def* **by** *auto*
qed

declaration ‹
Partial-Function-MR.init
sum-bot
(*fn* (*mt*, *t-to-ss*, *mtT*, *msT*, *t-to-sTs*) =>
list-comb (*Const* (@{*const-name sum-bot-map*}, *t-to-sTs* −−−> *mtT* −−> *msT*), *t-to-ss*) $ *mt*)
(*fn* (*commonTs*, *argTs*) => *Type* (@{*type-name sum-bot*}, *commonTs* @ *argTs*))

```
  (fn mT => Term.dest-Type mT |> #2 |> (fn [err, res] => ([err], [res])))
  @{thms sum-bot.map-comp}
  @{thms sum-bot.map-ident}
›

end
```

5 Monadic Parser Combinators

```
theory Parser-Monad
imports
  Error-Monad
  Show.Show
begin


abbreviation (input) tab ≡ CHR 0x09
abbreviation (input) carriage-return ≡ CHR 0x0D
abbreviation (input) wspace ≡ [CHR '' '', CHR ''↩'', tab, carriage-return]

definition trim :: string ⇒ string
  where trim = dropWhile (λc. c ∈ set wspace)

lemma trim:
  ∃ w. s = w @ trim s
  by (unfold trim-def) (metis takeWhile-dropWhile-id)
```

A parser takes a list of tokes and returns either an error message or a result together with the remaining tokens.

```
type-synonym
  ('t, 'a) gen-parser = 't list ⇒ string + ('a × 't list)

type-synonym
  'a parser = (char, 'a) gen-parser
```

5.1 Monad-Setup for Parsers

```
definition return :: 'a ⇒ ('t, 'a) gen-parser
where
  return x = (λts. Error-Monad.return (x, ts))

definition error :: string ⇒ ('t, 'a) gen-parser
where
  error e = (λ-. Error-Monad.error e)

definition bind :: ('t, 'a) gen-parser ⇒ ('a ⇒ ('t, 'b) gen-parser) ⇒ ('t, 'b)
gen-parser
where
  bind m f ts = do {
```

```
    (x, ts') ← m ts;
    f x ts'
  }

adhoc-overloading
  Monad-Syntax.bind ⇌ bind

lemma bind-cong [fundef-cong]:
  fixes m1 :: ('t, 'a) gen-parser
  assumes m1 ts2 = m2 ts2
    and ⋀ y ts. m2 ts2 = Inr (y, ts) ⟹ f1 y ts = f2 y ts
    and ts1 = ts2
  shows ((m1 ⤜ f1) ts1) = ((m2 ⤜ f2) ts2)
  using assms unfolding bind-def by (cases m1 ts1) auto

definition update-tokens :: ('t list ⇒ 't list) ⇒ ('t, 't list) gen-parser
where
  update-tokens f ts = Error-Monad.return (ts, f ts)

definition get-tokens :: ('t, 't list) gen-parser
where
  get-tokens = update-tokens (λx. x)

definition set-tokens :: 't list ⇒ ('t, unit) gen-parser
where
  [code-unfold]: set-tokens ts = update-tokens (λ-. ts) ≫ return ()

definition err-expecting :: string ⇒ ('t::show, 'a) gen-parser
where
  err-expecting msg ts = Error-Monad.error
    (''expecting '' @ msg @ '', but found: '' @ shows-quote (shows (take 30 ts)) [])

fun eoi :: ('t :: show, unit) gen-parser
where
  eoi [] = Error-Monad.return ((), []) |
  eoi ts = err-expecting ''end of input'' ts

fun exactly-aux :: string ⇒ string ⇒ string ⇒ string parser
where
  exactly-aux s i (x # xs) (y # ys) =
    (if x = y then exactly-aux s i xs ys
    else err-expecting (''''' @ s @ ''''') i) |
  exactly-aux s i [] xs = Error-Monad.return (s, trim xs) |
  exactly-aux s i (x # xs) [] = err-expecting (''''' @ s @ ''''') i

fun oneof-aux :: string list ⇒ string list ⇒ string parser
where
  oneof-aux allowed (x # xs) ts =
    (if map snd (zip x ts) = x then Error-Monad.return (x, trim (List.drop (length
```

```
x) ts))
    else oneof-aux allowed xs ts) |
  oneof-aux allowed [] ts = err-expecting (''one of '' @ shows-list allowed []) ts

definition is-parser :: 'a parser ⇒ bool where
  is-parser p ⟷ (∀ s r x. p s = Inr (x, r) ⟶ length s ≥ length r)

lemma is-parserI [intro]:
  assumes ⋀s r x. p s = Inr (x, r) ⟹ length s ≥ length r
  shows is-parser p
  using assms unfolding is-parser-def by blast

lemma is-parserE [elim]:
  assumes is-parser p
    and (⋀s r x. p s = Inr (x, r) ⟹ length s ≥ length r) ⟹ P
  shows P
  using assms by (auto simp: is-parser-def)

lemma is-parser-length:
  assumes is-parser p and p s = Inr (x, r)
  shows length s ≥ length r
  using assms by blast
```

A *consuming parser* (cparser for short) consumes at least one token of input.

```
definition is-cparser :: 'a parser ⇒ bool
where
  is-cparser p ⟷ (∀ s r x. p s = Inr (x, r) ⟶ length s > length r)

lemma is-cparserI [intro]:
  assumes ⋀s r x. p s = Inr (x, r) ⟹ length s > length r
  shows is-cparser p
  using assms unfolding is-cparser-def by blast

lemma is-cparserE [elim]:
  assumes is-cparser p
    and (⋀s r x. p s = Inr (x, r) ⟹ length s > length r) ⟹ P
  shows P
  using assms by (auto simp: is-cparser-def)

lemma is-cparser-length:
  assumes is-cparser p and p s = Inr (x, r)
  shows length s > length r
  using assms by blast

lemma is-parser-bind [intro, simp]:
  assumes p: is-parser p and q: ⋀ x. is-parser (q x)
  shows is-parser (p ⤜ q)
proof
  fix s r x
```

```
  assume (p ⤜ q) s = Inr (x, r)
  then obtain y t
    where P: p s = Inr (y, t) and Q: q y t = Inr (x, r)
    unfolding bind-def by (cases p s) auto
  have length r ≤ length t using q and Q by (auto simp: is-parser-def)
  also have ... ≤ length s using p and P by (auto simp: is-parser-def)
  finally show length r ≤ length s .
qed

definition oneof :: string list ⇒ string parser
where
  oneof xs = oneof-aux xs xs

lemma oneof-result:
  assumes oneof xs s = Inr (y, r)
  shows ∃ w. s = y @ w @ r ∧ y ∈ set xs
proof −
  {
    fix ys
    assume oneof-aux ys xs s = Inr (y,r)
    hence ∃ w. s = y @ w @ r ∧ y ∈ set xs
    proof (induct xs)
      case Nil thus ?case by (simp add: err-expecting-def)
    next
      case (Cons z zs)
      thus ?case
      proof (cases map snd (zip z s) = z)
        case False with Cons show ?thesis by simp
      next
        case True
        hence s: s = z @ drop (length z) s
        proof (induct z arbitrary: s)
          case (Cons a zz)
          thus ?case
            by (cases s, auto)
        qed simp
        from trim[of drop (length z) s] obtain w where drop (length z) s = w @
trim (drop (length z) s) by blast
        with s have s: s = z @ w @ trim (drop (length z) s) by auto
        from True Cons(2) have yz: y = z and r: r = trim (drop (length y) s) by
auto
        show ?thesis
          by (simp add: yz r, rule exI, rule s)
      qed
    qed
  }
  from this [OF assms [unfolded oneof-def]]
    show ?thesis .
qed
```

```
definition exactly :: string ⇒ string parser
where
  exactly s x = exactly-aux s x s x

lemma exactly-result:
  assumes exactly x s = Inr (y, r)
  shows ∃ w. s = x @ w @ r ∧ y = x
proof −
  {
    fix a b
    assume exactly-aux a b x s = Inr (y,r)
    hence ∃ w. s = x @ w @ r ∧ y = a
    proof (induct x arbitrary: s)
      case Nil
      thus ?case using trim[of s] by auto
    next
      case (Cons c xs) note xs = this
      show ?case
      proof (cases s)
        case Nil
        with xs show ?thesis by (auto simp: err-expecting-def)
      next
        case (Cons d ss)
        note xs = xs[unfolded Cons]
        from xs(2) have exactly-aux a b xs ss = Inr (y, r) ∧ c = d
          by (cases c = d) (auto simp: err-expecting-def)
        hence res: exactly-aux a b xs ss = Inr (y, r) and c: c = d by auto
        from xs(1)[OF res]
        show ?thesis unfolding Cons c by auto
      qed
    qed
  }
  from this[OF assms [unfolded exactly-def]] show ?thesis .
qed

hide-const oneof-aux exactly-aux

lemma oneof-length:
  assumes oneof xs s = Inr (y, r)
  shows length s ≥ length y + length r ∧ y ∈ set xs
proof −
  from oneof-result [OF assms]
  obtain w where s = y @ w @ r ∧ y ∈ set xs ..
  thus ?thesis  by auto
qed

lemma is-parser-oneof [intro]:
  is-parser (oneof ts)
```

proof
  fix *s r x*
  assume *oneof ts s = Inr (x ,r)*
  from *oneof-length* [*OF this*] **show** *length s ≥ length r* **by** *auto*
qed

lemma *is-cparser-oneof* [*intro*, *simp*]:
  assumes ∀ *x*∈*set ts*. *length x ≥ 1*
  shows *is-cparser* (*oneof ts*)
proof
  fix *s r x*
  assume *oneof ts s = Inr (x ,r)*
  from *oneof-length* [*OF this*] *assms*
    show *length s > length r* **by** *auto*
qed

lemma *exactly-length*:
  assumes *exactly x s = Inr* (*y*, *r*)
  shows *length s ≥ length x + length r*
proof −
  from *exactly-result* [*OF assms*]
  obtain *w* **where** *s = x* @ *w* @ *r* **by** *auto*
  thus *?thesis* **by** *auto*
qed

lemma *is-parser-exactly* [*intro*]:
  is-parser (*exactly xs*)
proof
  fix *s r x*
  assume *exactly xs s = Inr (x ,r)*
  from *exactly-length* [*OF this*]
    show *length s ≥ length r* **by** *auto*
qed

lemma *is-cparser-exactly* [*intro*]:
  assumes *length xs ≥ 1*
  shows *is-cparser* (*exactly xs*)
proof
  fix *s r x*
  assume *exactly xs s = Inr* (*x*, *r*)
  from *exactly-length* [*OF this*]
    show *length s > length r* **using** *assms* **by** *auto*
qed

fun *many* :: (*char* ⇒ *bool*) ⇒ (*char list*) *parser*
where
  many P (*t* # *ts*) =
    (*if P t then do* {
      (*rs*, *ts′*) ← *many P ts*;

```
    Error-Monad.return (t # rs, ts')
  } else Error-Monad.return ([], t # ts)) |
 many P [] = Error-Monad.return ([], [])

lemma is-parser-many [intro]:
 is-parser (many P)
proof
 fix s r x
 assume many P s = Inr (x, r)
 thus length r ≤ length s
 proof (induct s arbitrary: x r)
  case (Cons t ts)
  thus ?case by (cases P t, cases many P ts) force+
 qed simp
qed

definition manyof :: char list ⇒ (char list) parser
where
 [code-unfold]: manyof cs = many (λc. c ∈ set cs)

lemma is-parser-manyof [intro]:
 is-parser (manyof cs)
 unfolding manyof-def by blast

definition spaces :: unit parser
where
 [code-unfold]: spaces = manyof wspace ≫ return ()

lemma is-parser-return [intro]:
 is-parser (return x)
 by (auto simp: is-parser-def return-def)

lemma is-parser-error [intro]:
 is-parser (error x)
 by (auto simp: is-parser-def error-def)

lemma is-parser-If [intro!]:
 assumes is-parser p and is-parser q
 shows is-parser (if b then p else q)
 using assms by (cases b) auto

lemma is-parser-Let [intro!]:
 assumes is-parser (f y)
 shows is-parser (let x = y in f x)
 using assms by auto

lemma is-parser-spaces [intro]:
 is-parser spaces
 unfolding spaces-def by blast
```

```
fun scan-upto :: string ⇒ string parser
where
  scan-upto end (t # ts) =
    (if map snd (zip end (t # ts)) = end then do {
       Error-Monad.return (end, List.drop (length end) (t # ts))
    } else do {
      (res, ts') ← scan-upto end ts;
      Error-Monad.return (t # res, ts')
    }) |
  scan-upto end [] = Error-Monad.error (''did not find end−marker '' @ shows-quote
(shows end) [])

lemma scan-upto-length:
  assumes scan-upto end s = Inr (y, r)
  shows length s ≥ length end + length r
using assms
proof (induct s arbitrary: y r)
  case (Cons t ts)
  show ?case
  proof (cases map snd (zip end (t # ts)) = end)
    case True
    then obtain tss where tss: tss = t # ts and map: map snd (zip end tss) =
end by auto
    from map have len: length tss ≥ length end
    proof (induct end arbitrary: tss)
      case (Cons e en)
      thus ?case by (cases tss, auto)
    qed simp
    from True tss Cons(2) have y: y = end and r: r = List.drop (length end) tss
by auto
    show ?thesis by (simp only: tss[symmetric], simp add: y r, auto simp: len)
  next
    case False
    with Cons obtain res ts'
      where scan-upto end ts = Inr (res,ts') by (cases scan-upto end ts) (auto)
    from Cons(1)[OF this] Cons(2) False this show ?thesis by auto
  qed
qed simp

lemma is-parser-scan-upto [intro]:
  is-parser (scan-upto end)
  unfolding is-parser-def using scan-upto-length [of end] by force

lemma is-cparser-scan-upto [intro]:
  is-cparser (scan-upto (e # end))
  unfolding is-cparser-def using scan-upto-length [of e # end] by force

end
```

6 More material on parsing

theory *Misc*
  imports *Main*
begin

definition *span* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* × '*a list*
  where [*code-abbrev*, *simp*]: *span P xs* = (*takeWhile P xs*, *dropWhile P xs*)

lemma *span-code* [*code*]:
  span P [] = ([], [])
  span P (*x* # *xs*) =
    (*if P x then let* (*ys*, *zs*) = *span P xs in* (*x* # *ys*, *zs*) *else* ([], *x* # *xs*))
  by *simp-all*

definition *splitter* :: *char list* ⇒ *string* ⇒ *string* × *string*
where
  splitter cs s = *span* (λ*c*. *c* ∈ *set cs*) *s*

end