

Certification-Monads*

Christian Sternagel and René Thiemann

February 6, 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 these 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	6
3	A Special Error Monad for Certification with Informative Error Messages	8
4	A Sum Type with Bottom Element	11
4.1	Setup for Partial Functions	11
4.2	Monad Setup	11
4.3	Connection to <i>Partial-Function-MR.Partial-Function-MR</i> . .	13
5	Monadic Parser Combinators	14
5.1	Monad-Setup for Parsers	14
6	More material on parsing	19

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

syntax
  -replace-error :: 'a  $\Rightarrow$  'b  $\Rightarrow$  'a (infixl  $\langle$ <? $\rangle$  61)

syntax-consts
  -replace-error  $\Rightarrow$  update-error

translations
  m  $\langle$ ? e  $\rightarrow$  m  $\langle$ +? ( $\lambda$ -. 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  $\Rightarrow$  f x | Inl e  $\Rightarrow$  Inl e)
```

```
adhoc-overloading
  Monad-Syntax.bind  $\Rightarrow$  bind
```

```
abbreviation (input) return  $\equiv$  Inr
abbreviation (input) error  $\equiv$  Inl
abbreviation (input) run  $\equiv$  projr
```

2.1 Monad Laws

```
lemma return-bind [simp]:
  (return x  $\gg$  f) = f x
   $\langle$ proof $\rangle$ 
```

lemma *bind-return* [*simp*]:

$(m \gg= \text{return}) = m$
 $\langle \text{proof} \rangle$

lemma *error-bind* [*simp*]:

$(\text{error } e \gg= f) = \text{error } e$
 $\langle \text{proof} \rangle$

lemma *bind-assoc* [*simp*]:

fixes $m :: 'a + 'b$
shows $((m \gg= f) \gg= g) = (m \gg= (\lambda x. f x \gg= g))$
 $\langle \text{proof} \rangle$

lemma *bind-cong* [*fundef-cog*]:

fixes $m1\ m2 :: 'e + 'a$
and $f1\ f2 :: 'a \Rightarrow 'e + 'b$
assumes $m1 = m2$
and $\bigwedge y. m2 = \text{Inr } y \Rightarrow f1\ y = f2\ y$
shows $(m1 \gg= f1) = (m2 \gg= f2)$
 $\langle \text{proof} \rangle$

definition *catch-error* :: $'e + 'a \Rightarrow ('e \Rightarrow 'f + 'a) \Rightarrow 'f + 'a$

where

catch-def: $\text{catch-error } m\ f = (\text{case } m \text{ of } \text{Inl } e \Rightarrow f\ e \mid \text{Inr } x \Rightarrow \text{Inr } x)$

adhoc-overloading

Error-Syntax.catch $\hat{=} \text{catch-error}$

lemma *catch-splits*:

$P (\text{try } m \text{ catch } f) \longleftrightarrow (\forall e. m = \text{Inl } e \longrightarrow P (f\ e)) \wedge (\forall x. m = \text{Inr } x \longrightarrow P (\text{Inr } x))$
 $P (\text{try } m \text{ catch } f) \longleftrightarrow (\neg ((\exists e. m = \text{Inl } e \wedge \neg P (f\ e)) \vee (\exists x. m = \text{Inr } x \wedge \neg P (\text{Inr } x))))$
 $\langle \text{proof} \rangle$

abbreviation *update-error* :: $'e + 'a \Rightarrow ('e \Rightarrow 'f) \Rightarrow 'f + 'a$

where

update-error $m\ f \equiv \text{try } m \text{ catch } (\lambda x. \text{error } (f\ x))$

adhoc-overloading

Error-Syntax.update-error $\hat{=} \text{update-error}$

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

$(\text{try } \text{return } x \text{ catch } f) = \text{return } x \langle \text{proof} \rangle$

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

$(\text{try } \text{error } e \text{ catch } f) = f\ e \langle \text{proof} \rangle$

lemma *update-error-return* [*simp*]:
 $(m <+? c = \text{return } x) \longleftrightarrow (m = \text{return } x)$
 ⟨*proof*⟩

definition *isOk* $m \longleftrightarrow (\text{case } m \text{ of } \text{Inl } e \Rightarrow \text{False} \mid \text{Inr } x \Rightarrow \text{True})$

lemma *isOk-E* [*elim*]:
assumes *isOk* m
obtains x **where** $m = \text{return } x$
 ⟨*proof*⟩

lemma *isOk-I* [*simp, intro*]:
 $m = \text{return } x \Longrightarrow \text{isOk } m$
 ⟨*proof*⟩

lemma *isOk-iff*:
 $\text{isOk } m \longleftrightarrow (\exists x. m = \text{return } x)$
 ⟨*proof*⟩

lemma *isOk-error* [*simp*]:
 $\text{isOk } (\text{error } x) = \text{False}$
 ⟨*proof*⟩

lemma *isOk-bind* [*simp*]:
 $\text{isOk } (m \gg= f) \longleftrightarrow \text{isOk } m \wedge \text{isOk } (f (\text{run } m))$
 ⟨*proof*⟩

lemma *isOk-update-error* [*simp*]:
 $\text{isOk } (m <+? f) \longleftrightarrow \text{isOk } m$
 ⟨*proof*⟩

lemma *isOk-case-prod* [*simp*]:
 $\text{isOk } (\text{case } lr \text{ of } (l, r) \Rightarrow P \text{ } l \text{ } r) = (\text{case } lr \text{ of } (l, r) \Rightarrow \text{isOk } (P \text{ } l \text{ } r))$
 ⟨*proof*⟩

lemma *isOk-case-option* [*simp*]:
 $\text{isOk } (\text{case } x \text{ of } \text{None} \Rightarrow P \mid \text{Some } v \Rightarrow Q \text{ } v) = (\text{case } x \text{ of } \text{None} \Rightarrow \text{isOk } P \mid \text{Some } v \Rightarrow \text{isOk } (Q \text{ } v))$
 ⟨*proof*⟩

lemma *isOk-Let* [*simp*]:
 $\text{isOk } (\text{Let } s \text{ } f) = \text{isOk } (f \text{ } s)$
 ⟨*proof*⟩

lemma *run-bind* [*simp*]:
 $\text{isOk } m \Longrightarrow \text{run } (m \gg= f) = \text{run } (f (\text{run } m))$
 ⟨*proof*⟩

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

isOk $m \implies \text{run } (\text{try } m \text{ catch } f) = \text{run } m$
 ⟨proof⟩

fun *foldM* :: ('a ⇒ 'b ⇒ 'e + 'a) ⇒ 'a ⇒ 'b list ⇒ 'e + 'a
where
foldM f d [] = return d |
foldM f d ($x \# xs$) = do { $y \leftarrow f$ 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⟩

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))
 ⟨proof⟩

lemma *forallM-index* [fundef-cong]:
fixes c :: 'a ⇒ nat ⇒ 'e + unit
assumes $\bigwedge x i. x \in \text{set } xs \implies c$ x i = d x i
shows *forallM-index* c xs = *forallM-index* d xs
 ⟨proof⟩

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

lemma *forallM-fundef-cong* [fundef-cong]:
assumes $xs = ys \bigwedge x. x \in \text{set } ys \implies f$ x = g x
shows *forallM* f xs = *forallM* g ys
 ⟨proof⟩

lemma *isOk-forallM* [simp]:
 $isOk (forallM f xs) \longleftrightarrow (\forall x \in set\ xs. isOK (f x))$
 ⟨proof⟩

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 ⇒ 'e + unit) ⇒ 'a list ⇒ '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 $\bigwedge x. x \in set\ ys \implies f x = g x$
shows $existsM f xs = existsM g ys$
 ⟨proof⟩

lemma *isOk-existsM* [simp]:
 $isOk (existsM f xs) \longleftrightarrow (\exists x \in set\ xs. isOK (f x))$
 ⟨proof⟩

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

lemma *isOk-if-error* [simp]:
 $isOk (if b then error e else m) \longleftrightarrow \neg b \wedge isOK m$
 $isOk (if b then m else error e) \longleftrightarrow b \wedge isOK m$
 ⟨proof⟩

lemma *isOk-if*:
 $isOk (if b then x else y) \longleftrightarrow b \wedge isOK x \vee \neg b \wedge isOK y$
 ⟨proof⟩

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

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⟩

```

lemma *mapM-return*:

```

assumes mapM f xs = return ys
shows ys = map (run ∘ f) xs ∧ (∀ x ∈ set xs. ∀ e. f x ≠ error e)
⟨proof⟩

```

lemma *mapM-return-idx*:

```

assumes *: mapM f xs = Inr ys and i < length xs
shows ∃ y. f (xs ! i) = Inr y ∧ ys ! i = y
⟨proof⟩

```

lemma *mapM-cong [fundef-cong]*:

```

assumes xs = ys and ∧ x. x ∈ set ys ⇒ f x = g x
shows mapM f xs = mapM g ys
⟨proof⟩

```

lemma *bindE [elim]*:

```

assumes (p ≫= f) = return x
obtains y where p = return y and f y = return x
⟨proof⟩

```

lemma *then-return-eq [simp]*:

```

(p ≫= q) = return f ↔ isOK p ∧ q = return f
⟨proof⟩

```

fun *choice* :: ('e + 'a) list ⇒ 'e list + 'a

where

```

choice [] = error [] |
choice (x # xs) = (try x catch (λe. 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 (λx. run (f
x)) xs
⟨proof⟩

```

fun *firstM*

where

$firstM f [] = error []$
 $| firstM f (x \# xs) = (try f x \gg return x catch (\lambda e. firstM f xs <+? Cons e))$

lemma *firstM*:

$isOK (firstM f xs) \longleftrightarrow (\exists x \in set\ xs. isOK (f x))$
 $\langle proof \rangle$

lemma *firstM-return*:

assumes $firstM f xs = return y$
shows $isOK (f y) \wedge y \in set\ xs$
 $\langle proof \rangle$

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

$succeed \equiv return ()$

definition $check :: bool \Rightarrow 'e \Rightarrow 'e\ check$

where

$check\ b\ e = (if\ b\ then\ succeed\ else\ error\ e)$

lemma *isOK-check* [simp]:

$isOK (check\ b\ e) = b \langle proof \rangle$

lemma *isOK-check-catch* [simp]:

$isOK (try\ check\ b\ e\ catch\ f) \longleftrightarrow b \vee isOK (f\ e)$
 $\langle proof \rangle$

definition $check-return :: 'a\ check \Rightarrow 'b \Rightarrow 'a + 'b$

where

$check-return\ chk\ res = (chk \gg return\ res)$

lemma *check-return* [simp]:

$check-return\ chk\ res = return\ res' \longleftrightarrow isOK\ chk \wedge res' = res$

<proof>

lemma *[code-unfold]*:

$check\text{-}return\ chk\ res = (case\ chk\ of\ Inr\ - \Rightarrow Inr\ res\ | Inl\ e \Rightarrow Inl\ e)$
<proof>

abbreviation $check\text{-}allm :: ('a \Rightarrow 'e\ check) \Rightarrow 'a\ list \Rightarrow 'e\ check$

where

$check\text{-}allm\ f\ xs \equiv forallM\ f\ xs\ <+?\ snd$

abbreviation $check\text{-}exm :: ('a \Rightarrow 'e\ check) \Rightarrow 'a\ list \Rightarrow ('e\ list \Rightarrow 'e) \Rightarrow 'e\ check$

where

$check\text{-}exm\ f\ xs\ fld \equiv existsM\ f\ xs\ <+?\ fld$

lemma *isOK-check-allm*:

$isOK\ (check\text{-}allm\ f\ xs) \longleftrightarrow (\forall x \in set\ xs.\ isOK\ (f\ x))$
<proof>

abbreviation $check\text{-}allm\text{-}index :: ('a \Rightarrow nat \Rightarrow 'e\ check) \Rightarrow 'a\ list \Rightarrow 'e\ check$

where

$check\text{-}allm\text{-}index\ f\ xs \equiv forallM\text{-}index\ f\ xs\ <+?\ snd$

abbreviation $check\text{-}all :: ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ check$

where

$check\text{-}all\ f\ xs \equiv check\text{-}allm\ (\lambda x.\ if\ f\ x\ then\ succeed\ else\ error\ x)\ xs$

abbreviation $check\text{-}all\text{-}index :: ('a \Rightarrow nat \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow ('a \times nat)\ check$

where

$check\text{-}all\text{-}index\ f\ xs \equiv check\text{-}allm\text{-}index\ (\lambda x\ i.\ if\ f\ x\ i\ then\ succeed\ else\ error\ (x,\ i))\ xs$

lemma *isOK-check-all-index [simp]*:

$isOK\ (check\text{-}all\text{-}index\ f\ xs) \longleftrightarrow (\forall i < length\ xs.\ f\ (xs\ !\ i)\ i)$
<proof>

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

definition

$check\text{-}allm\text{-}gen\text{-}index ::$

$('a \Rightarrow nat \Rightarrow nat) \Rightarrow ('a \Rightarrow nat \Rightarrow 'e\ check) \Rightarrow nat \Rightarrow 'a\ list \Rightarrow 'e\ check$

where

$check\text{-}allm\text{-}gen\text{-}index\ g\ f\ n\ xs = snd\ (foldl\ (\lambda(i,\ m)\ x.\ (g\ x\ i,\ m \gg f\ x\ i))\ (n,\ succeed)\ xs)$

lemma *foldl-error*:

$snd\ (foldl\ (\lambda(i,\ m)\ x.\ (g\ x\ i,\ m \gg f\ x\ i))\ (n,\ error\ e)\ xs) = error\ e$
<proof>

lemma *isOK-check-allm-gen-index [simp]*:

assumes $isOK\ (check\text{-}allm\text{-}gen\text{-}index\ g\ f\ n\ xs)$

shows $\forall x \in \text{set } xs. \exists i. \text{isOK } (f \ x \ i)$
 ⟨proof⟩

lemma *check-allm-gen-index* [fundef-cong]:
fixes $f :: 'a \Rightarrow \text{nat} \Rightarrow 'e \text{ check}$
assumes $\bigwedge x \ n. x \in \text{set } xs \Longrightarrow g \ x \ n = g' \ x \ n$
and $\bigwedge x \ n. x \in \text{set } xs \Longrightarrow f \ x \ n = f' \ x \ n$
shows $\text{check-allm-gen-index } g \ f \ n \ xs = \text{check-allm-gen-index } g' \ f' \ n \ xs$
 ⟨proof⟩

definition *check-subseteq* :: $'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ check}$
where
 $\text{check-subseteq } xs \ ys = \text{check-all } (\lambda x. x \in \text{set } ys) \ xs$

lemma *isOK-check-subseteq* [simp]:
 $\text{isOK } (\text{check-subseteq } xs \ ys) \longleftrightarrow \text{set } xs \subseteq \text{set } ys$
 ⟨proof⟩

definition *check-same-set* :: $'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ check}$
where
 $\text{check-same-set } xs \ ys = (\text{check-subseteq } xs \ ys \gg \text{check-subseteq } ys \ xs)$

lemma *isOK-check-same-set* [simp]:
 $\text{isOK } (\text{check-same-set } xs \ ys) \longleftrightarrow \text{set } xs = \text{set } ys$
 ⟨proof⟩

definition *check-disjoint* :: $'a \text{ list} \Rightarrow 'a \text{ list} \Rightarrow 'a \text{ check}$
where
 $\text{check-disjoint } xs \ ys = \text{check-all } (\lambda x. x \notin \text{set } ys) \ xs$

lemma *isOK-check-disjoint* [simp]:
 $\text{isOK } (\text{check-disjoint } xs \ ys) \longleftrightarrow \text{set } xs \cap \text{set } ys = \{\}$
 ⟨proof⟩

definition *check-all-combinations* :: $('a \Rightarrow 'a \Rightarrow 'b \text{ check}) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ check}$
where
 $\text{check-all-combinations } c \ xs = \text{check-allm } (\lambda x. \text{check-allm } (c \ x) \ xs) \ xs$

lemma *isOK-check-all-combinations* [simp]:
 $\text{isOK } (\text{check-all-combinations } c \ xs) \longleftrightarrow (\forall x \in \text{set } xs. \forall y \in \text{set } xs. \text{isOK } (c \ x \ y))$
 ⟨proof⟩

fun *check-pairwise* :: $('a \Rightarrow 'a \Rightarrow 'b \text{ check}) \Rightarrow 'a \text{ list} \Rightarrow 'b \text{ check}$
where
 $\text{check-pairwise } c \ [] = \text{succeed}$ |
 $\text{check-pairwise } c \ (x \ \# \ xs) = (\text{check-allm } (c \ x) \ xs \gg \text{check-pairwise } c \ xs)$

lemma *pairwise-aux*:
 $(\forall j < \text{length } (x \ \# \ xs). \forall i < j. P ((x \ \# \ xs) ! i) ((x \ \# \ xs) ! j))$

$$= ((\forall j < \text{length } xs. P \ x \ (xs \ ! \ j)) \wedge (\forall j < \text{length } xs. \forall i < j. P \ (xs \ ! \ i) \ (xs \ ! \ j)))$$
(is ?C = (?A ∧ ?B))
 <proof>

lemma *isOK-check-pairwise* [simp]:

$$isOK \ (check\text{-pairwise } c \ xs) \longleftrightarrow (\forall j < \text{length } xs. \forall i < j. isOK \ (c \ (xs \ ! \ i) \ (xs \ ! \ j)))$$
 <proof>

abbreviation *check-exists* :: ('a ⇒ bool) ⇒ 'a list ⇒ ('a list) check
where

check-exists f xs ≡ check-exm (λx. if f x then succeed else error [x]) xs concat

lemma *isOK-choice* [simp]:

$$isOK \ (choice \ []) \longleftrightarrow False$$

$$isOK \ (choice \ (x \ \# \ xs)) \longleftrightarrow isOK \ x \ \vee \ isOK \ (choice \ xs)$$
 <proof>

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]: $isOK \ (or\text{-ok } a \ b) \longleftrightarrow isOK \ a \ \vee \ isOK \ b$ <proof>

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* (**infix** <+_⊥> 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

<proof>

<ML>

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** $\bigwedge x. ys = \text{Right } x \implies f x = g x$

shows *bind xs f* = *bind ys g*

<proof>

abbreviation *mono-sum-bot* :: (('a ⇒ ('e +_⊥ 'b)) ⇒ 'f +_⊥ 'c) ⇒ *bool*

where

mono-sum-bot ≡ *monotone (fun-ord sum-bot-ord) sum-bot-ord*

lemma *bind-mono* [*partial-function-mono*]:

assumes *mf*: *mono-sum-bot B* **and** *mg*: $\bigwedge y. \text{mono-sum-bot } (\lambda f. C y f)$

shows *mono-sum-bot* ($\lambda f. \text{bind } (B f) (\lambda y. C y f)$)

<proof>

adhoc-overloading

Monad-Syntax.bind ⇐ *bind*

hide-const (open) *bind*

fun *catch-error* :: 'e +_⊥ 'a ⇒ ('e ⇒ ('f +_⊥ 'a)) ⇒ 'f +_⊥ '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 ⇐ *catch-error*

lemma *catch-mono* [*partial-function-mono*]:

assumes *mf*: *mono-sum-bot B* **and** *mg*: $\bigwedge y. \text{mono-sum-bot } (\lambda f. C y f)$

shows *mono-sum-bot* ($\lambda f. \text{try } (B f) \text{ catch } (\lambda y. C y f)$)

<proof>

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** $\bigwedge x. x \in \text{set } ys \implies f x = g x$
shows *map-sum-bot* f xs = *map-sum-bot* g ys
⟨*proof*⟩

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 $\bigwedge y. y \in \text{set } B \implies \text{mono-sum-bot } (C y)$
shows *mono-sum-bot* ($\lambda f. \text{map-sum-bot } (\lambda y. C y f) B$)
⟨*proof*⟩

abbreviation *update-error* :: 'e +_⊥ 'a ⇒ ('e ⇒ 'f) ⇒ 'f +_⊥ 'a
where
update-error r f ≡ *try* r *catch* ($\lambda e. \text{error } (f e)$)

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)
⟨*proof*⟩

lemma [*code*]:
(*try* (*sumbot* a) *catch* f) = (case a of Inl b ⇒ f b | Inr a ⇒ *sumbot* (Inr a))
⟨*proof*⟩

lemma [*code*]: *Right* x = *sumbot* (Inr x) ⟨*proof*⟩

lemma [*code*]: *Left* x = *sumbot* (Inl x) ⟨*proof*⟩

lemma [code]: *return* $x = \text{sumbot } (\text{Inr } x)$ $\langle \text{proof} \rangle$

lemma [code]: *error* $x = \text{sumbot } (\text{Inl } x)$ $\langle \text{proof} \rangle$

lemma [code]:
case-sum-bot $f g h (\text{sumbot } p) = \text{case-sum } g h p$
 $\langle \text{proof} \rangle$

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* $(\lambda f. \text{sum-bot-map } h (B f))$
 $\langle \text{proof} \rangle$

$\langle ML \rangle$

end

5 Monadic Parser Combinators

theory *Parser-Monad*

imports

Error-Monad

Show.Show

begin

abbreviation (*input*) *tab* $\equiv \text{CHR } 0x09$

abbreviation (*input*) *carriage-return* $\equiv \text{CHR } 0x0D$

abbreviation (*input*) *wspace* $\equiv [\text{CHR } " ", \text{CHR } "\u2190"], \text{tab}, \text{carriage-return}]$

definition *trim* $:: \text{string} \Rightarrow \text{string}$

where *trim* = *dropWhile* $(\lambda c. c \in \text{set } \text{wspace})$

lemma *trim*:

$\exists w. s = w @ \text{trim } s$

$\langle \text{proof} \rangle$

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

type-synonym

$(t, 'a) \text{ gen-parser} = t \text{ list} \Rightarrow \text{string} + ('a \times t \text{ list})$

type-synonym

$'a \text{ parser} = (\text{char}, 'a) \text{ gen-parser}$

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)

<proof>

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 *eof* :: ('t :: show, unit) gen-parser

where

eof [] = Error-Monad.return ((), []) |

eof 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*
⟨*proof*⟩

lemma *is-parserE* [*elim*]:

assumes *is-parser* *p*
 and (∧ *s* *r* *x*. *p* *s* = *Inr* (*x*, *r*) ⇒ *length* *s* ≥ *length* *r*) ⇒ *P*
shows *P*
⟨*proof*⟩

lemma *is-parser-length*:

assumes *is-parser* *p* **and** *p* *s* = *Inr* (*x*, *r*)
shows *length* *s* ≥ *length* *r*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *is-cparserE* [*elim*]:

assumes *is-cparser* *p*
 and (∧ *s* *r* *x*. *p* *s* = *Inr* (*x*, *r*) ⇒ *length* *s* > *length* *r*) ⇒ *P*

shows P
 $\langle proof \rangle$

lemma *is-cparser-length*:
assumes *is-cparser* p **and** $p\ s = \text{Inr}\ (x, r)$
shows $\text{length}\ s > \text{length}\ r$
 $\langle proof \rangle$

lemma *is-parser-bind* [*intro*, *simp*]:
assumes p : *is-parser* p **and** q : $\bigwedge x. \text{is-parser}\ (q\ x)$
shows *is-parser* $(p \gg= q)$
 $\langle proof \rangle$

definition *oneof* :: *string list* \Rightarrow *string parser*
where
 $\text{oneof}\ xs = \text{oneof-aux}\ xs\ xs$

lemma *oneof-result*:
assumes $\text{oneof}\ xs\ s = \text{Inr}\ (y, r)$
shows $\exists w. s = y\ @\ w\ @\ r \wedge y \in \text{set}\ xs$
 $\langle proof \rangle$

definition *exactly* :: *string* \Rightarrow *string parser*
where
 $\text{exactly}\ s\ x = \text{exactly-aux}\ s\ x\ s\ x$

lemma *exactly-result*:
assumes $\text{exactly}\ x\ s = \text{Inr}\ (y, r)$
shows $\exists w. s = x\ @\ w\ @\ r \wedge y = x$
 $\langle proof \rangle$

hide-const *oneof-aux exactly-aux*

lemma *oneof-length*:
assumes $\text{oneof}\ xs\ s = \text{Inr}\ (y, r)$
shows $\text{length}\ s \geq \text{length}\ y + \text{length}\ r \wedge y \in \text{set}\ xs$
 $\langle proof \rangle$

lemma *is-parser-oneof* [*intro*]:
is-parser $(\text{oneof}\ ts)$
 $\langle proof \rangle$

lemma *is-cparser-oneof* [*intro*, *simp*]:
assumes $\forall x \in \text{set}\ ts. \text{length}\ x \geq 1$
shows *is-cparser* $(\text{oneof}\ ts)$
 $\langle proof \rangle$

lemma *exactly-length*:
assumes $\text{exactly}\ x\ s = \text{Inr}\ (y, r)$

shows $length\ s \geq length\ x + length\ r$
<proof>

lemma *is-parser-exactly* [intro]:
is-parser (exactly *xs*)
<proof>

lemma *is-cparser-exactly* [intro]:
assumes $length\ xs \geq 1$
shows *is-cparser* (exactly *xs*)
<proof>

fun *many* :: (char \Rightarrow bool) \Rightarrow (char list) parser
where
many *P* (*t* # *ts*) =
 (if *P* *t* then do {
 (*rs*, *ts'*) \leftarrow *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>

definition *manyof* :: char list \Rightarrow (char list) parser
where
[code-unfold]: *manyof* *cs* = *many* ($\lambda c. c \in set\ cs$)

lemma *is-parser-manyof* [intro]:
is-parser (*manyof* *cs*)
<proof>

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

lemma *is-parser-return* [intro]:
is-parser (return *x*)
<proof>

lemma *is-parser-error* [intro]:
is-parser (error *x*)
<proof>

lemma *is-parser-If* [intro!]:
assumes *is-parser* *p* **and** *is-parser* *q*
shows *is-parser* (if *b* then *p* else *q*)
<proof>

```

lemma is-parser-Let [intro!]:
  assumes is-parser (f y)
  shows is-parser (let x = y in f x)
  ⟨proof⟩

lemma is-parser-spaces [intro]:
  is-parser spaces
  ⟨proof⟩

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
  ⟨proof⟩

lemma is-parser-scan-upto [intro]:
  is-parser (scan-upto end)
  ⟨proof⟩

lemma is-cparser-scan-upto [intro]:
  is-cparser (scan-upto (e # end))
  ⟨proof⟩

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 $(ys, zs) = \text{span } P xs$ in $(x \# ys, zs)$ else $([], x \# xs)$)
<proof>

definition *splitter* :: *char list* \Rightarrow *string* \Rightarrow *string* \times *string*

where

splitter cs s = span ($\lambda c. c \in \text{set } cs$) s

end