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 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 *Partial-Function-MR.Partial-Function-MR* . . 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 :: $'a \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'c$ (‹(*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 $\rightleftharpoons$ *update-error*

translations
$m <? e \rightharpoonup m <+? (\lambda\text{-}.\ e)$

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 $\rightleftharpoons$ *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*
⟨*proof*⟩

lemma *bind-return* [*simp*]:
(*m* ⤜ *return*) = *m*
⟨*proof*⟩

lemma *error-bind* [*simp*]:
(*error e* ⤜ *f*) = *error e*
⟨*proof*⟩

lemma *bind-assoc* [*simp*]:
fixes *m* :: *'a* + *'b*
shows ((*m* ⤜ *f*) ⤜ *g*) = (*m* ⤜ (λ*x*. *f x* ⤜ *g*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*))))
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *catch-error* [*simp*]:
(*try error e catch f*) = *f e* ⟨*proof*⟩

lemma *update-error-return* [*simp*]:
(*m* <+? *c* = *return x*) ⟷ (*m* = *return x*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *isOK-I* [*simp*, *intro*]:
m = *return x* ⟹ *isOK m*
⟨*proof*⟩

lemma *isOK-iff*:
isOK m ⟷ (∃ *x*. *m* = *return x*)
⟨*proof*⟩

lemma *isOK-error* [*simp*]:
isOK (*error x*) = *False*
⟨*proof*⟩

lemma *isOK-bind* [*simp*]:
isOK (*m* ⤜ *f*) ⟷ *isOK m* ∧ *isOK* (*f* (*run m*))
⟨*proof*⟩

lemma *isOK-update-error* [*simp*]:
isOK (*m* <+? *f*) ⟷ *isOK m*
⟨*proof*⟩

lemma *isOK-case-prod* [*simp*]:
isOK (*case lr of* (*l*, *r*) ⇒ *P l r*) = (*case lr of* (*l*, *r*) ⇒ *isOK* (*P l r*))
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *isOK-Let* [*simp*]:
isOK (*Let s f*) = *isOK* (*f s*)
⟨*proof*⟩

lemma *run-bind* [*simp*]:
isOK m ⟹ *run* (*m* ⤜ *f*) = *run* (*f* (*run m*))
⟨*proof*⟩

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

isOK m $\Longrightarrow$ *run* (*try m catch f*) = *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* ← *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* ∈ *set xs* $\Longrightarrow$ *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* ∈ *set ys* $\Longrightarrow$ *f x* = *g x*
shows *forallM f xs* = *forallM g ys*
⟨*proof*⟩

lemma *isOK-forallM* [*simp*]:
isOK (*forallM f xs*) ⟷ (∀ *x* ∈ *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* (λ*e*. *existsM f xs* <+? *Cons e*))

lemma *existsM-cong* [*fundef-cong*]:
assumes *xs* = *ys*
and ⋀*x*. *x* ∈ *set ys* ⟹ *f x* = *g x*
shows *existsM f xs* = *existsM g ys*
⟨*proof*⟩

lemma *isOK-existsM* [*simp*]:
isOK (*existsM f xs*) ⟷ (∃ *x*∈*set xs*. *isOK* (*f x*))
⟨*proof*⟩

lemma *is-OK-if-return* [*simp*]:
isOK (*if b then return x else m*) ⟷ *b* ∨ *isOK m*
isOK (*if b then m else return x*) ⟷ ¬ *b* ∨ *isOK m*
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *isOK-if*:
isOK (*if b then x else y*) ⟷ *b* ∧ *isOK x* ∨ ¬ *b* ∧ *isOK y*
⟨*proof*⟩

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⟩

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* ≫ *return x catch* (λ*e*. *firstM f xs* <+? *Cons e*))

lemma *firstM*:
isOK (*firstM f xs*) ⟷ (∃ *x*∈*set xs*. *isOK* (*f x*))
⟨*proof*⟩

lemma *firstM-return*:
assumes *firstM f xs* = *return y*
shows *isOK* (*f y*) ∧ *y* ∈ *set xs*
⟨*proof*⟩

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* ⟨*proof*⟩

lemma *isOK-check-catch* [*simp*]:
isOK (*try check b e catch f*) ⟷ *b* ∨ *isOK* (*f e*)
⟨*proof*⟩

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*

⟨*proof*⟩

lemma [*code-unfold*]:
check-return chk res = (*case chk of Inr* - ⇒ *Inr res* | *Inl e* ⇒ *Inl e*)
⟨*proof*⟩

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*))
⟨*proof*⟩

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*)
⟨*proof*⟩

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*
⟨*proof*⟩

lemma *isOK-check-allm-gen-index* [*simp*]:
assumes *isOK* (*check-allm-gen-index g f n xs*)

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

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

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

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

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

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

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

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

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

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

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

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

$= ((\forall j{<}\mathit{length}\ \mathit{xs}.\ P\ x\ (\mathit{xs}\ !\ j)) \wedge (\forall j{<}\mathit{length}\ \mathit{xs}.\ \forall i{<}j.\ P\ (\mathit{xs}\ !\ i)\ (\mathit{xs}\ !\ j)))$
(**is** $?C = (?A \wedge ?B)$)
⟨*proof*⟩

lemma *isOK-check-pairwise* [*simp*]:
$\mathit{isOK}\ (\mathit{check\text{-}pairwise}\ c\ \mathit{xs}) \longleftrightarrow (\forall j{<}\mathit{length}\ \mathit{xs}.\ \forall i{<}j.\ \mathit{isOK}\ (c\ (\mathit{xs}\ !\ i)\ (\mathit{xs}\ !\ j)))$
⟨*proof*⟩

abbreviation *check-exists* :: $('a \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{list} \Rightarrow ('a\ \mathit{list})\ \mathit{check}$
where
$\mathit{check\text{-}exists}\ f\ \mathit{xs} \equiv \mathit{check\text{-}exm}\ (\lambda x.\ \mathit{if}\ f\ x\ \mathit{then}\ \mathit{succeed}\ \mathit{else}\ \mathit{error}\ [x])\ \mathit{xs}\ \mathit{concat}$

lemma *isOK-choice* [*simp*]:
$\mathit{isOK}\ (\mathit{choice}\ []) \longleftrightarrow \mathit{False}$
$\mathit{isOK}\ (\mathit{choice}\ (x\ \#\ \mathit{xs})) \longleftrightarrow \mathit{isOK}\ x \vee \mathit{isOK}\ (\mathit{choice}\ \mathit{xs})$
⟨*proof*⟩

fun *or-ok* :: $'a\ \mathit{check} \Rightarrow 'a\ \mathit{check} \Rightarrow 'a\ \mathit{check}$ **where**
$\mathit{or\text{-}ok}\ (\mathit{Inl}\ a)\ b = b\ |$
$\mathit{or\text{-}ok}\ (\mathit{Inr}\ a)\ b = \mathit{Inr}\ a$

lemma *or-is-or*[*simp*]: $\mathit{isOK}\ (\mathit{or\text{-}ok}\ a\ b) \longleftrightarrow \mathit{isOK}\ a \vee \mathit{isOK}\ b$ ⟨*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* (**infixr** ‹$+_\perp$› *10*) = *Bottom* | *Left* $'e$ | *Right* $'a$
for *map*: *sum-bot-map*

4.1 Setup for Partial Functions

abbreviation *sum-bot-ord* :: $'e +_\perp 'a \Rightarrow 'e +_\perp 'a \Rightarrow \mathit{bool}$
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* :: $'e +_\bot 'a \Rightarrow ('a \Rightarrow ('e +_\bot 'b)) \Rightarrow 'e +_\bot 'b$
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*
⟨*proof*⟩

abbreviation *mono-sum-bot* :: $(('a \Rightarrow ('e +_\bot 'b)) \Rightarrow 'f +_\bot 'c) \Rightarrow$ *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*: $\bigwedge y.$ *mono-sum-bot* ($\lambda f.$ *C y f*)
shows *mono-sum-bot* ($\lambda f.$ *bind* (*B f*) ($\lambda y.$ *C y f*))
⟨*proof*⟩

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

lemma *catch-mono* [*partial-function-mono*]:
assumes *mf*: *mono-sum-bot B* **and** *mg*: $\bigwedge y.$ *mono-sum-bot* ($\lambda f.$ *C y f*)
shows *mono-sum-bot* ($\lambda f.$ *try* (*B f*) *catch* ($\lambda y.$ *C y f*))
⟨*proof*⟩

definition *error* :: $'e \Rightarrow 'e +_\bot 'a$
where
[*simp*]: *error x* = *Left x*

definition *return* :: $'a \Rightarrow 'e +_\bot 'a$
where
[*simp*]: *return x* = *Right x*

fun *map-sum-bot* :: $('a \Rightarrow ('e +_\bot 'b)) \Rightarrow 'a\ list \Rightarrow 'e +_\bot 'b\ list$
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** $\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x$
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* :: $'a \Rightarrow ('b \Rightarrow ('e +_\bot 'c)) \Rightarrow 'e +_\bot 'd$
assumes $\bigwedge y.\ y \in set\ B \Longrightarrow mono\text{-}sum\text{-}bot\ (C\ y)$
shows *mono-sum-bot* ($\lambda f.$ *map-sum-bot* ($\lambda y.$ *C y f*) *B*)
⟨*proof*⟩

abbreviation *update-error* :: $'e +_\bot 'a \Rightarrow ('e \Rightarrow 'f) \Rightarrow 'f +_\bot 'a$
where
update-error r f ≡ *try r catch* (λ *e. error* (*f e*))

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

fun *sumbot* :: $'e + 'a \Rightarrow 'e +_\bot 'a$
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 = sumbot (Inr x)* ⟨*proof*⟩

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

lemma [*code*]:
case-sum-bot f g h (sumbot p) = case-sum g h p
⟨*proof*⟩

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*⟩

⟨*ML*⟩

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* (λ*c*. *c* ∈ *set wspace*)

lemma *trim*:
∃ *w*. *s* = *w* @ *trim s*
⟨*proof*⟩

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 = (λ*ts*. *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* (λ*x*. *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*
⟨*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
⟨*proof*⟩

lemma *is-cparser-length*:
assumes *is-cparser p* **and** $p\ s = Inr\ (x,\ r)$
shows *length s* > *length r*
⟨*proof*⟩

lemma *is-parser-bind* [*intro*, *simp*]:
assumes *p*: *is-parser p* **and** *q*: $\bigwedge x.$ *is-parser* $(q\ x)$
shows *is-parser* $(p \gg\!= q)$
⟨*proof*⟩

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

lemma *oneof-result*:
assumes *oneof xs s* = *Inr* $(y,\ r)$
shows $\exists\ w.\ s = y\ @\ w\ @\ r \wedge y \in set\ xs$
⟨*proof*⟩

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 $\exists\ w.\ s = x\ @\ w\ @\ r \wedge y = x$
⟨*proof*⟩

hide-const *oneof-aux exactly-aux*

lemma *oneof-length*:
assumes *oneof xs s* = *Inr* $(y,\ r)$
shows *length s* ≥ *length y* + *length r* ∧ $y \in set\ xs$
⟨*proof*⟩

lemma *is-parser-oneof* [*intro*]:
is-parser (*oneof ts*)
⟨*proof*⟩

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

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

shows *length s* ≥ *length x* + *length r*
⟨*proof*⟩

lemma *is-parser-exactly* [*intro*]:
is-parser (*exactly xs*)
⟨*proof*⟩

lemma *is-cparser-exactly* [*intro*]:
assumes *length xs* ≥ *1*
shows *is-cparser* (*exactly xs*)
⟨*proof*⟩

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*⟩

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

lemma *is-parser-manyof* [*intro*]:
is-parser (*manyof cs*)
⟨*proof*⟩

definition *spaces* :: *unit parser*
where
[*code-unfold*]: *spaces* = *manyof wspace* ≫ *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) = *span* P xs *in* (x # ys, zs) *else* ([], x # xs))
⟨*proof*⟩

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

end