Xml*

Christian Sternagel and René Thiemann

September 13, 2023

Abstract

This entry provides an "XML library" for Isabelle/HOL. This includes parsing and pretty printing of XML trees as well as combinators for transforming XML trees into arbitrary user-defined data. The main contribution of this entry is an interface (fit for code generation) that allows for communication between verified programs formalized in Isabelle/HOL and the outside world via XML. This library was developed as part of the IsaFoR/CeTA project to which we refer for examples of its usage.

Contents

1 Parsing and Printing XML Documents

theory *Xml*
imports
Certification-Monads.Parser-Monad
HOL−Library.Char-ord
HOL−Library.Code-Abstract-Char
begin

datatype *xml* =
— node-name, attributes, child-nodes
XML string (*string* × *string*) *list xml list* |
XML-text string

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

```
datatype xmldoc =
  — header, body
  XMLDOC string list (root-node: xml)

fun tag :: xml ⇒ string where
  tag (XML name - - ) = name |
  tag (XML-text -) = []
hide-const (open) tag

fun children :: xml ⇒ xml list where
  children (XML - - cs) = cs |
  children (XML-text -) = []
hide-const (open) children

fun num-children :: xml ⇒ nat where
  num-children (XML - - cs) = length cs |
  num-children (XML-text -) = 0
hide-const (open) num-children
```

1.1 Printing of XML Nodes and Documents

```
instantiation xml :: show
begin

definition shows-attr :: string × string ⇒ shows
where
  shows-attr av = shows (fst av) o shows-string (''='' @ snd av @ '''')

definition shows-attrs :: (string × string) list ⇒ shows
where
  shows-attrs as = foldr (λa. '' '' +#+ shows-attr a) as

fun shows-XML-indent :: string ⇒ nat ⇒ xml ⇒ shows
where
  shows-XML-indent ind i (XML n a c) =
    (''↩'' +#+ ind +#+ ''<'' +#+ shows n +@+ shows-attrs a +@+
      (if c = [] then shows-string ''/>''
      else (
        ''>'' +#+
         foldr (shows-XML-indent (replicate i (CHR '' '') @ ind) i) c +@+ ''↩''
+#+ ind +#+
        ''</'' +#+ shows n +@+ shows-string ''>''))) |
  shows-XML-indent ind i (XML-text t) = shows-string t

definition shows-prec (d::nat) xml = shows-XML-indent '''' 2 xml

definition shows-list (xs :: xml list) = showsp-list shows-prec 0 xs
```

lemma *shows-attr-append*:
  $(s$ +#+ *shows-attr av*$)$ $(r$ @ $t) = (s$ +#+ *shows-attr av*$)$ r @ t
  ⟨*proof*⟩

lemma *shows-attrs-append* [*show-law-simps*]:
  shows-attrs as $(r$ @ $s) =$ *shows-attrs as* r @ s
  ⟨*proof*⟩

lemma *append-xml′*:
  shows-XML-indent ind i xml $(r$ @ $s) =$ *shows-XML-indent ind i xml* r @ s
  ⟨*proof*⟩

lemma *shows-prec-xml-append* [*show-law-simps*]:
  shows-prec d (*xml*::*xml*) $(r$ @ $s) =$ *shows-prec d xml* r @ s
  ⟨*proof*⟩

instance
  ⟨*proof*⟩

end

instantiation *xmldoc* :: *show*
begin

fun *shows-xmldoc*
where
  shows-xmldoc (*XMLDOC h x*) = *shows-lines h o shows-nl o shows x*

definition *shows-prec* (*d*::*nat*) *doc* = *shows-xmldoc doc*
definition *shows-list* (*xs* :: *xmldoc list*) = *showsp-list shows-prec 0 xs*

lemma *shows-prec-xmldoc-append* [*show-law-simps*]:
  shows-prec d (*x*::*xmldoc*) $(r$ @ $s) =$ *shows-prec d x* r @ s
  ⟨*proof*⟩

instance
  ⟨*proof*⟩

end

1.2 XML-Parsing

definition *parse-text* :: *string option parser*
where
  parse-text = *do* {
    ts ← *many* ((≠) *CHR* ″<″);
    let text = *trim ts*;
    if text = [] *then return None*
    else return (*Some* (*List.rev* (*trim* (*List.rev text*))))

}

lemma *is-parser-parse-text* [*intro*]:
is-parser parse-text
⟨*proof*⟩

lemma *parse-text-consumes*:
assumes ∗: $ts \neq []$ $hd\ ts \neq CHR\ ''<''$
and *parse*: $parse\text{-}text\ ts = Inr\ (t,\ ts')$
shows $length\ ts' < length\ ts$
⟨*proof*⟩

definition *parse-attribute-value* :: *string parser*
where
parse-attribute-value = *do* {
exactly [*CHR* '''];
$v \leftarrow many\ ((\neq)\ CHR\ ''')$;
exactly [*CHR* '''];
return v
}

lemma *is-parser-parse-attribute-value* [*intro*]:
is-parser parse-attribute-value
⟨*proof*⟩

A list of characters that are considered to be "letters" for tag-names.

definition *letters* :: *char list*
where
letters = ''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ-0123456789&;:−''

definition *is-letter* :: *char* ⇒ *bool*
where
$is\text{-}letter\ c \longleftrightarrow c \in set\ letters$

lemma *is-letter-pre-code*:
$is\text{-}letter\ c \longleftrightarrow$
$CHR\ ''a'' \leq c \wedge c \leq CHR\ ''z'' \vee$
$CHR\ ''A'' \leq c \wedge c \leq CHR\ ''Z'' \vee$
$CHR\ ''0'' \leq c \wedge c \leq CHR\ ''9'' \vee$
$c \in set$ ''-&;:−''
⟨*proof*⟩

definition *many-letters* :: *string parser*
where
[*simp*]: *many-letters* = *manyof letters*

lemma *many-letters* [*code*, *code-unfold*]:
many-letters = *many is-letter*
⟨*proof*⟩

```
definition parse-name :: string parser
where
  parse-name s = (do {
    n ← many-letters;
    spaces;
    if n = [] then
      error (''expected letter '' @ letters @ '' but first symbol is '' @ take 1 s @ '''')
    else return n
  }) s

lemma is-parser-parse-name [intro]:
  is-parser parse-name
⟨proof⟩

function (sequential) parse-attributes :: (string × string) list parser
where
  parse-attributes [] = Error-Monad.return ([], []) |
  parse-attributes (c # s) =
    (if c ∈ set ''/>'' then Error-Monad.return ([], c # s)
    else (do {
      k ← parse-name;
      exactly ''='';
      v ← parse-attribute-value;
      atts ← parse-attributes;
      return ((k, v) # atts)
    }) (c # s))
  ⟨proof⟩

termination parse-attributes
⟨proof⟩

lemma is-parser-parse-attributes [intro]:
  is-parser parse-attributes
⟨proof⟩

context notes [[function-internals]]
begin

function parse-nodes :: xml list parser
where
  parse-nodes ts =
    (if ts = [] ∨ take 2 ts = ''</'' then return [] ts
    else if hd ts ≠ CHR ''<'' then (do {
      t ← parse-text;
      ns ← parse-nodes;
      return (XML-text (the t) # ns)
    }) ts
    else (do {
```

```
    exactly ''<'';
    n ← parse-name;
    atts ← parse-attributes;
    e ← oneof [''/>'', ''>''];
    (λ ts'.
      if e = ''/>'' then (do {
        cs ← parse-nodes;
        return (XML n atts [] # cs)
      }) ts' else (do {
        cs ← parse-nodes;
        exactly ''</'';
        exactly n;
        exactly ''>'';
        ns ← parse-nodes;
        return (XML n atts cs # ns)
      }) ts')
  }) ts)
⟨proof⟩
```

end

lemma *parse-nodes-help*:
parse-nodes-dom s ∧ (∀ *x r*. *parse-nodes s* = *Inr* (*x*, *r*) ⟶ *length r* ≤ *length s*)
(**is** *?prop s*)
⟨*proof*⟩

termination *parse-nodes* ⟨*proof*⟩

lemma *parse-nodes* [*intro*]:
is-parser parse-nodes
⟨*proof*⟩

A more efficient variant of *oneof* [''/>'', ''>''].

```
fun oneof-closed :: string parser
where
  oneof-closed (x # xs) =
    (if x = CHR ''>'' then Error-Monad.return (''>'', trim xs)
    else if x = CHR ''/'' ∧ (case xs of [] ⇒ False | y # ys ⇒ y = CHR ''>'') then
      Error-Monad.return (''/>'', trim (tl xs))
    else err-expecting (''one of [/>, >]'') (x # xs)) |
  oneof-closed xs = err-expecting (''one of [/>, >]'') xs
```

lemma *oneof-closed*:
oneof [''/>'', ''>''] = *oneof-closed* (**is** *?l* = *?r*)
⟨*proof*⟩

lemma *If-removal*:
(λ *e x*. *if b e then f e x else g e x*) = (λ *e*. *if b e then f e else g e*)
⟨*proof*⟩

```
declare parse-nodes.simps [unfolded oneof-closed,
  unfolded If-removal [of λ e. e = ''/>''], code]

definition parse-node :: xml parser
where
  parse-node = do {
    exactly ''<'';
    n ← parse-name;
    atts ← parse-attributes;
    e ← oneof [''/>'', ''>''];
    if e = ''/>'' then return (XML n atts [])
    else do {
      cs ← parse-nodes;
      exactly ''</'';
      exactly n;
      exactly ''>'';
      return (XML n atts cs)
    }
  }

declare parse-node-def [unfolded oneof-closed, code]

function parse-header :: string list parser
where
  parse-header ts =
    (if take 2 (trim ts) = ''<?'' then (do {
      h ← scan-upto ''?>'';
      hs ← parse-header;
      return (h # hs)
    }) ts else (do {
      spaces;
      return []
    }) ts)
  ⟨proof⟩

termination parse-header
⟨proof⟩


definition comment-error = Code.abort (STR ''comment not terminated'') (λ -.
'''')
definition comment-error-hyphen = Code.abort (STR ''double hyphen within com-
ment'') (λ -. '''')

fun rc-aux where rc-aux False (c # cs) =
    (if c = CHR ''<'' ∧ take 3 cs = ''!--'' then rc-aux True (drop 3 cs)
    else c # rc-aux False cs) |
  rc-aux True (c # cs) =
```

(*if* c = *CHR* ''−'' ∧ *take 1 cs* = ''−'' *then*
if take 2 cs = ''−'' *then comment-error else if take 2 cs* = ''−>'' *then rc-aux False* (*drop 2 cs*)
else comment-error-hyphen
else rc-aux True cs) |
rc-aux False [] = [] |
rc-aux True [] = *comment-error*

definition *remove-comments xs* = *rc-aux False xs*

definition *rc-open-1 xs* = *rc-aux False xs*
definition *rc-open-2 xs* = *rc-aux False* (*CHR* ''<'' # *xs*)
definition *rc-open-3 xs* = *rc-aux False* (*CHR* ''<'' # *CHR* ''!'' # *xs*)
definition *rc-open-4 xs* = *rc-aux False* (*CHR* ''<'' # *CHR* ''!'' # *CHR* ''−'' # *xs*)
definition *rc-close-1 xs* = *rc-aux True xs*
definition *rc-close-2 xs* = *rc-aux True* (*CHR* ''−'' # *xs*)
definition *rc-close-3 xs* = *rc-aux True* (*CHR* ''−'' # *CHR* ''−'' # *xs*)

lemma *remove-comments-code*[*code*]: *remove-comments xs* = *rc-open-1 xs*
⟨*proof*⟩

lemma *char-eq-via-integer-eq*: $c = d \longleftrightarrow$ *integer-of-char* c = *integer-of-char* d
⟨*proof*⟩

lemma *integer-of-char-simps*[*simp*]:
integer-of-char (*CHR* ''<'') = *60*
integer-of-char (*CHR* ''>'') = *62*
integer-of-char (*CHR* ''/'') = *47*
integer-of-char (*CHR* ''!'') = *33*
integer-of-char (*CHR* ''−'') = *45*
⟨*proof*⟩

lemma *rc-open-close-simp*[*code*]:
rc-open-1 (c # *cs*) = (*if integer-of-char* c = *60 then rc-open-2 cs else* c # *rc-open-1 cs*)
rc-open-1 [] = []
rc-open-2 (c # *cs*) = (*let ic* = *integer-of-char* c *in if ic* = *33 then rc-open-3 cs else if ic* = *60 then* c # *rc-open-2 cs else CHR* ''<'' # c # *rc-open-1 cs*)
rc-open-2 [] = ''<''
rc-open-3 (c # *cs*) = (*let ic* = *integer-of-char* c *in if ic* = *45 then rc-open-4 cs else if ic* = *60 then* c # *CHR* ''!'' # *rc-open-2 cs else CHR* ''<'' # *CHR* ''!'' # c # *rc-open-1 cs*)
rc-open-3 [] = ''<!''
rc-open-4 (c # *cs*) = (*let ic* = *integer-of-char* c *in if ic* = *45 then rc-close-1 cs else if ic* = *60 then* c # *CHR* ''!'' # *CHR* ''−'' # *rc-open-2 cs else CHR* ''<'' # *CHR* ''!'' # *CHR* ''−'' # c # *rc-open-1 cs*)
rc-open-4 [] = ''<!−''

rc-close-1 (*c* # *cs*) = (*if integer-of-char c* = *45 then rc-close-2 cs else rc-close-1 cs*)
rc-close-1 [] = *comment-error*
rc-close-2 (*c* # *cs*) = (*if integer-of-char c* = *45 then rc-close-3 cs else rc-close-1 cs*)
rc-close-2 [] = *comment-error*
rc-close-3 (*c* # *cs*) = (*if integer-of-char c* = *62 then rc-open-1 cs else comment-error-hyphen*)
rc-close-3 [] = *comment-error*
⟨*proof*⟩

definition *parse-doc* :: *xmldoc parser*
where
parse-doc = *do* {
update-tokens remove-comments;
h ← *parse-header*;
xml ← *parse-node*;
eoi;
return (*XMLDOC h xml*)
}

definition *doc-of-string* :: *string* ⇒ *string* + *xmldoc*
where
doc-of-string s = *do* {
(*doc*, -) ← *parse-doc s*;
Error-Monad.return doc
}

1.3 More efficient code equations

lemma *trim-code*[*code*]:
trim = *dropWhile* (λ *c*. *let ci* = *integer-of-char c*
in if ci ≥ *34 then False else ci* = *32* ∨ *ci* = *10* ∨ *ci* = *9* ∨ *ci* = *13*)
⟨*proof*⟩

fun *parse-text-main* :: *string* ⇒ *string* ⇒ *string* × *string* **where**
parse-text-main [] *res* = (''', *rev* (*trim res*))
| *parse-text-main* (*c* # *cs*) *res* = (*if c* = *CHR* ''<'' *then* (*c* # *cs*, *rev* (*trim res*))
else parse-text-main cs (*c* # *res*))

definition *parse-text-impl cs* = (*case parse-text-main* (*trim cs*) ''' *of*
(*rem*, *txt*) ⇒ *if txt* = [] *then Inr* (*None*, *rem*) *else Inr* (*Some txt*, *rem*))

lemma *parse-text-main*: *parse-text-main xs ys* =
(*dropWhile* ((≠) *CHR* ''<'') *xs*, *rev* (*trim* (*rev* (*takeWhile* ((≠) *CHR* ''<'') *xs*) @ *ys*)))
⟨*proof*⟩

lemma *many-take-drop*: *many f xs* = *Inr* (*takeWhile f xs*, *dropWhile f xs*)
⟨*proof*⟩

lemma *trim-takeWhile-inside*: *trim* (*takeWhile* ((≠) *CHR* ''<'') *cs*) = *takeWhile* ((≠) *CHR* ''<'') (*trim cs*)
⟨*proof*⟩

lemma *trim-dropWhile-inside*: *dropWhile* ((≠) *CHR* ''<'') *cs* = *dropWhile* ((≠) *CHR* ''<'') (*trim cs*)
⟨*proof*⟩

declare [[*code drop*: *parse-text*]]

lemma *parse-text-code*[*code*]: *parse-text cs* = *parse-text-impl cs*
⟨*proof*⟩

declare [[*code drop*: *parse-text-main*]]

lemma *parse-text-main-code*[*code*]:
parse-text-main [] *res* = ('''', *rev* (*trim res*))
parse-text-main (*c* # *cs*) *res* = (*if integer-of-char c* = *60 then* (*c* # *cs*, *rev* (*trim res*))
else parse-text-main cs (*c* # *res*))
⟨*proof*⟩

lemma *exactly-head*: *exactly* [*c*] (*c* # *cs*) = *Inr* ([*c*],*trim cs*)
⟨*proof*⟩

lemma *take-1-test*: (*case cs of* [] ⇒ *False* | *c* # *x* ⇒ *c* = *CHR* ''/'') = (*take 1 cs* = ''/'')
⟨*proof*⟩

definition *exactly-close* = *exactly* ''>''
definition *exactly-end* = *exactly* ''</''

lemma *exactly-close-code*[*code*]:
exactly-close [] = *err-expecting* (''>'') []
exactly-close (*c* # *cs*) = (*if integer-of-char c* = *62 then Inr* (''>'', *trim cs*) *else err-expecting* (''>'') (*c* # *cs*))
⟨*proof*⟩

lemma *exactly-end-code*[*code*]:
exactly-end [] = *err-expecting* (''</'') []
exactly-end [*c*] = *err-expecting* (''</'') [*c*]
exactly-end (*c* # *d* # *cs*) = (*if integer-of-char c* = *60* ∧ *integer-of-char d* = *47 then Inr* (''</'', *trim cs*)
else err-expecting (''</'') (*c* # *d* # *cs*))

⟨*proof*⟩

fun *oneof-closed-combined* :: *'a parser* ⇒ *'a parser* ⇒ *'a parser* **where**
oneof-closed-combined p q (x # xs) =
(*if x* = *CHR ''>''* *then q (trim xs)*
else if x = *CHR ''/''* ∧ (*case xs of* [] ⇒ *False* | *y # ys* ⇒ *y* = *CHR ''>''*) *then*
p (trim (tl xs))
else err-expecting (''one of [/>, >]'') (x # xs)) |
oneof-closed-combined p q xs = *err-expecting (''one of [/>, >]'') xs*

lemma *oneof-closed-combined*: *oneof-closed-combined p q* = (*oneof-closed* ≫= (λ*e*. *if e* = *''/>''* *then p else q*)) (**is** *?l* = *?r*)
⟨*proof*⟩

declare [[*code drop*: *oneof-closed-combined*]]

lemma *oneof-closed-combined-code*[*code*]:
oneof-closed-combined p q [] = *err-expecting (''one of [/>, >]'') ''''*
oneof-closed-combined p q (x # xs) = (*let xi* = *integer-of-char x in*
(*if xi* = *62 then q (trim xs)*
else (*if xi* = *47 then*
(*case xs of* [] ⇒ *err-expecting (''one of [/>, >]'') (x # xs)*
| *y # ys* ⇒ *if integer-of-char y* = *62 then p (trim ys)*
else err-expecting (''one of [/>, >]'') (x # xs))
else err-expecting (''one of [/>, >]'') (x # xs)))))
⟨*proof*⟩

lemmas *parse-nodes-current-code*
= *parse-nodes.simps*[*unfolded oneof-closed*, *unfolded If-removal* [*of* λ *e*. *e* = *''/>''*]]

lemma *parse-nodes-pre-code*:
parse-nodes (c # cs) =
(*if c* = *CHR ''<''* *then*
if (*case cs of* [] ⇒ *False* | *c* # - ⇒ *c* = *CHR ''/''*) *then Parser-Monad.return* [] (*c # cs*)
else (*parse-name* ≫=
(λ*n*. *parse-attributes* ≫=
(λ*atts*.
oneof-closed-combined (*parse-nodes* ≫= (λ*cs*. *Parser-Monad.return* (*XML n atts* [] # *cs*)))
(*parse-nodes* ≫=
(λ*cs*. *exactly-end* ≫=
(λ-. *exactly n* ≫=
(λ-. *exactly-close* ≫=
(λ-. *parse-nodes* ≫= (λ*ns*. *Parser-Monad.return* (*XML n atts cs* # *ns*)))))))))
(*trim cs*)
else (*parse-text* ≫= (λ*t*. *parse-nodes* ≫= (λ*ns*. *Parser-Monad.return* (*XML-text*

(*the t*) # *ns*)))) (*c* # *cs*))
⟨*proof*⟩

declare [[*code drop*: *parse-nodes*]]

lemma *parse-nodes-code*[*code*]:
parse-nodes [] = *Parser-Monad.return* [] ''''
parse-nodes (*c* # *cs*) =
(*if integer-of-char c* = *60 then*
if (*case cs of* [] ⇒ *False* | *d* # - ⇒ *d* = *CHR* ''/'') *then Parser-Monad.return* [] (*c* # *cs*)
else (*parse-name* ≫=
(λ*n*. *parse-attributes* ≫=
(λ*atts*.
oneof-closed-combined (*parse-nodes* ≫= (λ*cs*. *Parser-Monad.return* (*XML n atts* [] # *cs*)))
(*parse-nodes* ≫=
(λ*cs*. *exactly-end* ≫=
(λ-. *exactly n* ≫=
(λ-. *exactly-close* ≫=
(λ-. *parse-nodes* ≫= (λ*ns*. *Parser-Monad.return* (*XML n atts cs* # *ns*)))))))))
(*trim cs*)
else (*parse-text* ≫= (λ*t*. *parse-nodes* ≫= (λ*ns*. *Parser-Monad.return* (*XML-text* (*the t*) # *ns*)))) (*c* # *cs*))
⟨*proof*⟩

declare [[*code drop*: *parse-attributes*]]

lemma *parse-attributes-code*[*code*]:
parse-attributes [] = *Error-Monad.return* ([], [])
parse-attributes (*c* # *s*) = (*let ic* = *integer-of-char c in*
(*if ic* = *47* ∨ *ic* = *62 then Inr* ([], *c* # *s*)
else (*parse-name* ≫=
(λ*k*. *exactly* ''='' ≫= (λ-. *parse-attribute-value* ≫= (λ*v*. *parse-attributes* ≫= (λ*atts*. *Parser-Monad.return* ((*k*, *v*) # *atts*))))))
(*c* # *s*)))
⟨*proof*⟩

declare [[*code drop*: *is-letter*]]

lemma *is-letter-code*[*code*]: *is-letter c* = (*let ci* = *integer-of-char c in*
(*97* ≤ *ci* ∧ *ci* ≤ *122* ∨
65 ≤ *ci* ∧ *ci* ≤ *90* ∨
48 ≤ *ci* ∧ *ci* ≤ *59* ∨
ci = *95* ∨ *ci* = *38* ∨ *ci* = *45*))
⟨*proof*⟩

declare *spaces-def*[*code-unfold del*]

lemma *spaces-code*[*code*]:
  spaces cs = *Inr* (()*, trim cs*)
  ⟨*proof*⟩

declare *many-letters*[*code del*, *code-unfold del*]

fun *many-letters-main* **where**
  many-letters-main [] = ([], [])
| *many-letters-main* (*c* # *cs*) = (*if is-letter c then*
    case many-letters-main cs of (*ds*,*es*) ⇒ (*c* # *ds*, *es*)
    else ([], *c* # *cs*))

lemma *many-letters-code*[*code*]: *many-letters cs* = *Inr* (*many-letters-main cs*)
  ⟨*proof*⟩

lemma *parse-name-code*[*code*]:
  parse-name s = (*case many-letters-main s of*
    (*n*, *ts*) ⇒ *if n* = [] *then Inl*
      (''*expected letter* '' @ *letters* @ '' *but first symbol is* '' @ *take 1 s* @ '''')
    else Inr (*n*, *trim ts*))
  ⟨*proof*⟩

end

2 XML Transformers for Extracting Data from XML Nodes

theory *Xmlt*
imports
  Xml
  Certification-Monads.Strict-Sum
  HOL.Rat
begin

type-synonym
  tag = *string*

The type of transformers on xml nodes.

type-synonym
  'a xmlt = *xml* ⇒ *string* $+_\perp$ *'a*

definition *map* :: (*xml* ⇒ (*'e* $+_\perp$ *'a*)) ⇒ *xml list* ⇒ *'e* $+_\perp$ *'a list*
where
  [*code-unfold*]: *map* = *map-sum-bot*

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

fixes C :: *xml* $\Rightarrow$ ($'b \Rightarrow ('e +_\perp 'c)) \Rightarrow 'e +_\perp 'd$
assumes C: $\bigwedge y.\ y \in$ *set* $B \Longrightarrow$ *mono-sum-bot* ($C\ y$)
shows *mono-sum-bot* ($\lambda f.$ *map* ($\lambda y.\ C\ y\ f$) B)
⟨*proof*⟩

hide-const (**open**) *map*

fun *text* :: *tag* $\Rightarrow$ *string xmlt*
where
text tag (*XML n atts* [*XML-text t*]) =
(*if n* = *tag* $\wedge$ *atts* = [] *then return t*
else error (*concat*
["*could not extract text for* ", *tag*," *from* ", "$\boxed{\hookleftarrow}$", *show* (*XML n atts* [*XML-text t*])]))
| *text tag xml* = *error* (*concat* ["*could not extract text for* ", *tag*," *from* ", "$\boxed{\hookleftarrow}$", *show xml*])
hide-const (**open**) *text*

definition *bool-of-string* :: *string* $\Rightarrow$ *string* $+_\perp$ *bool*
where
bool-of-string s =
(*if s* = "*true*" *then return True*
else if s = "*false*" *then return False*
else error ("*cannot convert* " @ *s* @ " *into Boolean*"))

fun *bool* :: *tag* $\Rightarrow$ *bool xmlt*
where
bool tag node = *Xmlt.text tag node* $\gg=$ *bool-of-string*
hide-const (**open**) *bool*

definition *fail* :: *tag* $\Rightarrow$ $'a$ *xmlt*
where
fail tag xml =
error (*concat*
["*could not transform the following xml element* (*expected* ", *tag*, ")", "$\boxed{\hookleftarrow}$", *show xml*])
hide-const (**open**) *fail*

definition *guard* :: (*xml* $\Rightarrow$ *bool*) $\Rightarrow$ $'a$ *xmlt* $\Rightarrow$ $'a$ *xmlt* $\Rightarrow$ $'a$ *xmlt*
where
guard p p1 p2 x = (*if p x then p1 x else p2 x*)
hide-const (**open**) *guard*

lemma *guard-mono* [*partial-function-mono*]:
assumes *p1*: $\bigwedge y.$ *mono-sum-bot* (*p1 y*)
and *p2*: $\bigwedge y.$ *mono-sum-bot* (*p2 y*)
shows *mono-sum-bot* ($\lambda g.$ *Xmlt.guard p* ($\lambda y.$ *p1 y g*) ($\lambda y.$ *p2 y g*) *x*)
⟨*proof*⟩

```
fun leaf :: tag ⇒ 'a ⇒ 'a xmlt
where
  leaf tag x (XML name atts cs) =
    (if name = tag ∧ atts = [] ∧ cs = [] then return x
    else Xmlt.fail tag (XML name atts cs)) |
  leaf tag x xml = Xmlt.fail tag xml
hide-const (open) leaf

fun list1element :: 'a list ⇒ 'a option
where
  list1element [x] = Some x |
  list1element - = None

fun singleton :: tag ⇒ 'a xmlt ⇒ ('a ⇒ 'b) ⇒ 'b xmlt
where
  singleton tag p1 f xml =
    (case xml of
      XML name atts cs ⇒
      (if name = tag ∧ atts = [] then
        (case list1element cs of
          Some (cs1) ⇒ p1 cs1 ≫= return ∘ f
        | None ⇒ Xmlt.fail tag xml)
      else Xmlt.fail tag xml)
    | - ⇒ Xmlt.fail tag xml)
hide-const (open) singleton

lemma singleton-mono [partial-function-mono]:
  assumes p: ⋀y. mono-sum-bot (p1 y)
  shows mono-sum-bot (λg. Xmlt.singleton t (λy. p1 y g) f x)
    ⟨proof⟩

fun list2elements :: 'a list ⇒ ('a × 'a) option
where
  list2elements [x, y] = Some (x, y) |
  list2elements - = None

fun pair :: tag ⇒ 'a xmlt ⇒ 'b xmlt ⇒ ('a ⇒ 'b ⇒ 'c) ⇒ 'c xmlt
where
  pair tag p1 p2 f xml =
    (case xml of
      XML name atts cs ⇒
      (if name = tag ∧ atts = [] then
        (case list2elements cs of
          Some (cs1, cs2) ⇒
          do {
            a ← p1 cs1;
            b ← p2 cs2;
            return (f a b)
          }
```

```
      | None ⇒ Xmlt.fail tag xml)
    else Xmlt.fail tag xml)
  | - ⇒ Xmlt.fail tag xml)
hide-const (open) pair

lemma pair-mono [partial-function-mono]:
  assumes ⋀y. mono-sum-bot (p1 y)
    and ⋀y. mono-sum-bot (p2 y)
  shows mono-sum-bot (λg. Xmlt.pair t (λy. p1 y g) (λ y. p2 y g) f x)
  ⟨proof⟩

fun list3elements :: 'a list ⇒ ('a × 'a × 'a) option
where
  list3elements [x, y, z] = Some (x, y, z) |
  list3elements - = None

fun triple :: string ⇒ 'a xmlt ⇒ 'b xmlt ⇒ 'c xmlt ⇒ ('a ⇒ 'b ⇒ 'c ⇒ 'd) ⇒ 'd
xmlt
where
  triple tag p1 p2 p3 f xml = (case xml of XML name atts cs ⇒
    (if name = tag ∧ atts = [] then
      (case list3elements cs of
        Some (cs1, cs2, cs3) ⇒
        do {
          a ← p1 cs1;
          b ← p2 cs2;
          c ← p3 cs3;
          return (f a b c)
        }
      | None ⇒ Xmlt.fail tag xml)
    else Xmlt.fail tag xml)
  | - ⇒ Xmlt.fail tag xml)

lemma triple-mono [partial-function-mono]:
  assumes ⋀y. mono-sum-bot (p1 y)
    and ⋀y. mono-sum-bot (p2 y)
    and ⋀y. mono-sum-bot (p3 y)
  shows mono-sum-bot (λg. Xmlt.triple t (λy. p1 y g) (λ y. p2 y g) (λ y. p3 y g)
f x)
  ⟨proof⟩

fun list4elements :: 'a list ⇒ ('a × 'a × 'a × 'a) option
where
  list4elements [x, y, z, u] = Some (x, y, z, u) |
  list4elements - = None

fun
  tuple4 ::
    string ⇒ 'a xmlt ⇒ 'b xmlt ⇒ 'c xmlt ⇒ 'd xmlt ⇒ ('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e)
```

```
⇒ 'e xmlt
where
  tuple4 tag p1 p2 p3 p4 f xml =
    (case xml of
      XML name atts cs ⇒
        (if name = tag ∧ atts = [] then
          (case list4elements cs of
            Some (cs1, cs2, cs3, cs4) ⇒
            do {
              a ← p1 cs1;
              b ← p2 cs2;
              c ← p3 cs3;
              d ← p4 cs4;
              return (f a b c d)
            }
          | None ⇒ Xmlt.fail tag xml)
        else Xmlt.fail tag xml)
  | - ⇒ Xmlt.fail tag xml)
```

lemma *tuple4-mono* [*partial-function-mono*]:
assumes $\bigwedge$*y. mono-sum-bot (p1 y)*
and $\bigwedge$*y. mono-sum-bot (p2 y)*
and $\bigwedge$*y. mono-sum-bot (p3 y)*
and$\bigwedge$*y. mono-sum-bot (p4 y)*
shows *mono-sum-bot* (λ*g. Xmlt.tuple4 t* (λ*y. p1 y g*) (λ *y. p2 y g*) (λ *y. p3 y g*) (λ *y. p4 y g*) *f x*)
⟨*proof*⟩

```
fun list5elements :: 'a list ⇒ ('a × 'a × 'a × 'a × 'a) option
where
  list5elements [x, y, z, u, v] = Some (x, y, z, u, v) |
  list5elements - = None

fun
  tuple5 ::
    string ⇒ 'a xmlt ⇒ 'b xmlt ⇒ 'c xmlt ⇒ 'd xmlt ⇒ 'e xmlt ⇒
      ('a ⇒ 'b ⇒ 'c ⇒ 'd ⇒ 'e ⇒ 'f) ⇒ 'f xmlt
where
  tuple5 tag p1 p2 p3 p4 p5 f xml =
    (case xml of
      XML name atts cs ⇒
        (if name = tag ∧ atts = [] then
          (case list5elements cs of
            Some (cs1,cs2,cs3,cs4,cs5) ⇒
            do {
              a ← p1 cs1;
              b ← p2 cs2;
              c ← p3 cs3;
              d ← p4 cs4;
```

```
            e ← p5 cs5;
            return (f a b c d e)
          }
        | None ⇒ Xmlt.fail tag xml)
      else Xmlt.fail tag xml)
  | - ⇒ Xmlt.fail tag xml)
```

lemma *tuple5-mono* [*partial-function-mono*]:
assumes $\bigwedge y$. *mono-sum-bot* (*p1 y*)
and $\bigwedge y$. *mono-sum-bot* (*p2 y*)
and $\bigwedge y$. *mono-sum-bot* (*p3 y*)
and $\bigwedge y$. *mono-sum-bot* (*p4 y*)
and $\bigwedge y$. *mono-sum-bot* (*p5 y*)
shows *mono-sum-bot* (λg. *Xmlt.tuple5 t* (λy. *p1 y g*) (λ *y*. *p2 y g*) (λ *y*. *p3 y g*) (λ *y*. *p4 y g*) (λ *y*. *p5 y g*) *f x*)
⟨*proof*⟩

fun *list6elements* :: *'a list* ⇒ (*'a* × *'a* × *'a* × *'a* × *'a* × *'a*) *option*
where
list6elements [*x*, *y*, *z*, *u*, *v*, *w*] = *Some* (*x*, *y*, *z*, *u*, *v*, *w*) |
list6elements - = *None*

fun
tuple6 ::
string ⇒ *'a xmlt* ⇒ *'b xmlt* ⇒ *'c xmlt* ⇒ *'d xmlt* ⇒ *'e xmlt* ⇒ *'f xmlt* ⇒
(*'a* ⇒ *'b* ⇒ *'c* ⇒ *'d* ⇒ *'e* ⇒ *'f* ⇒ *'g*) ⇒ *'g xmlt*
where
tuple6 tag p1 p2 p3 p4 p5 p6 f xml =

```
  (case xml of
    XML name atts cs  ⇒
      (if name = tag ∧ atts = [] then
        (case list6elements cs of
          Some (cs1,cs2,cs3,cs4,cs5,cs6) ⇒
          do {
            a ← p1 cs1;
            b ← p2 cs2;
            c ← p3 cs3;
            d ← p4 cs4;
            e ← p5 cs5;
            ff ← p6 cs6;
            return (f a b c d e ff)
          }
        | None ⇒ Xmlt.fail tag xml)
      else Xmlt.fail tag xml)
  | - ⇒ Xmlt.fail tag xml)
```

lemma *tuple6-mono* [*partial-function-mono*]:
assumes $\bigwedge y$. *mono-sum-bot* (*p1 y*)
and $\bigwedge y$. *mono-sum-bot* (*p2 y*)

and $\bigwedge y.$ *mono-sum-bot* (*p3 y*)
and $\bigwedge y.$ *mono-sum-bot* (*p4 y*)
and $\bigwedge y.$ *mono-sum-bot* (*p5 y*)
and $\bigwedge y.$ *mono-sum-bot* (*p6 y*)
shows *mono-sum-bot* ($\lambda g.$ *Xmlt.tuple6 t* ($\lambda y.$ *p1 y g*) (λ *y. p2 y g*) (λ *y. p3 y g*) (λ *y. p4 y g*) (λ *y. p5 y g*) (λ *y. p6 y g*) *f x*)
⟨*proof*⟩

fun *optional* :: *tag* $\Rightarrow$ *'a xmlt* $\Rightarrow$ (*'a option* $\Rightarrow$ *'b*) $\Rightarrow$ *'b xmlt*
where
optional tag p1 f (*XML name atts cs*) =
(*let l* = *length cs in*
(*if name* = *tag* $\wedge$ *atts* = [] $\wedge$ *l* $\geq$ *0* $\wedge$ *l* $\leq$ *1 then do* {
if l = *1 then do* {
x1 $\leftarrow$ *p1* (*cs* ! *0*);
return (*f* (*Some x1*))
} *else return* (*f None*)
} *else Xmlt.fail tag* (*XML name atts cs*))) |
optional tag p1 f xml = *Xmlt.fail tag xml*

lemma *optional-mono* [*partial-function-mono*]:
assumes $\bigwedge y.$ *mono-sum-bot* (*p1 y*)
shows *mono-sum-bot* ($\lambda g.$ *Xmlt.optional t* ($\lambda y.$ *p1 y g*) *f x*)
⟨*proof*⟩

fun *xml1to2elements* :: *string* $\Rightarrow$ *'a xmlt* $\Rightarrow$ *'b xmlt* $\Rightarrow$ (*'a* $\Rightarrow$ *'b option* $\Rightarrow$ *'c*) $\Rightarrow$ *'c xmlt*
where
xml1to2elements tag p1 p2 f (*XML name atts cs*) = (
let l = *length cs in*
(*if name* = *tag* $\wedge$ *atts* = [] $\wedge$ *l* $\geq$ *1* $\wedge$ *l* $\leq$ *2*
then do {
x1 $\leftarrow$ *p1* (*cs* ! *0*);
(*if l* = *2*
then do {
x2 $\leftarrow$ *p2* (*cs* ! *1*);
return (*f x1* (*Some x2*))
} *else return* (*f x1 None*))
} *else Xmlt.fail tag* (*XML name atts cs*))) |
xml1to2elements tag p1 p2 f xml = *Xmlt.fail tag xml*

lemma *xml1to2elements-mono*[*partial-function-mono*]:
assumes *p1*: $\bigwedge y.$ *mono-sum-bot* (*p1 y*)
$\bigwedge y.$ *mono-sum-bot* (*p2 y*)
shows *mono-sum-bot* ($\lambda g.$ *xml1to2elements t* ($\lambda y.$ *p1 y g*) ($\lambda y.$ *p2 y g*) *f x*)
⟨*proof*⟩

Apply the first transformer to the first child-node, then check the second child-node, which is must be a Boolean. If the Boolean is true, then apply

the second transformer to the last child-node.

fun *xml2nd-choice* :: *tag* ⇒ *'a xmlt* ⇒ *tag* ⇒ *'b xmlt* ⇒ (*'a* ⇒ *'b option* ⇒ *'c*) ⇒ *'c xmlt*
where
  xml2nd-choice tag p1 cn p2 f (*XML name atts cs*) = (
    let l = *length cs in*
    (*if name* = *tag* ∧ *atts* = [] ∧ *l* ≥ *2 then do* {
      x1 ← *p1* (*cs* ! *0*);
      b ← *Xmlt.bool cn* (*cs* ! *1*);
      (*if b then do* {
        x2 ← *p2* (*cs* ! (*l* − *1*));
        return (*f x1* (*Some x2*))
      } *else return* (*f x1 None*))
    } *else Xmlt.fail tag* (*XML name atts cs*))) |
  xml2nd-choice tag p1 cn p2 f xml = *Xmlt.fail tag xml*

lemma *xml2nd-choice-mono* [*partial-function-mono*]:
  assumes *p1*: ⋀*y*. *mono-sum-bot* (*p1 y*)
            ⋀*y*. *mono-sum-bot* (*p2 y*)
  shows *mono-sum-bot* (λ*g*. *xml2nd-choice t* (λ*y*. *p1 y g*) *h* (λ*y*. *p2 y g*) *f x*)
  ⟨*proof*⟩

fun
  xml2to3elements ::
    string ⇒ *'a xmlt* ⇒ *'b xmlt* ⇒ *'c xmlt* ⇒ (*'a* ⇒ *'b* ⇒ *'c option* ⇒ *'d*) ⇒ *'d xmlt*
where
  xml2to3elements tag p1 p2 p3 f (*XML name atts cs*) = (
    let l = *length cs in*
    (*if name* = *tag* ∧ *atts* = [] ∧ *l* ≥ *2* ∧ *l* ≤ *3 then do* {
      x1 ← *p1* (*cs* ! *0*);
      x2 ← *p2* (*cs* ! *1*);
      (*if l* = *3 then do* {
        x3 ← *p3* (*cs* ! *2*);
        return (*f x1 x2* (*Some x3*))
      } *else return* (*f x1 x2 None*))
    } *else Xmlt.fail tag* (*XML name atts cs*))) |
  xml2to3elements tag p1 p2 p3 f xml = *Xmlt.fail tag xml*

lemma *xml2to3elements-mono* [*partial-function-mono*]:
  assumes *p1*: ⋀*y*. *mono-sum-bot* (*p1 y*)
            ⋀*y*. *mono-sum-bot* (*p2 y*)
            ⋀*y*. *mono-sum-bot* (*p3 y*)
  shows *mono-sum-bot* (λ*g*. *xml2to3elements t* (λ*y*. *p1 y g*) (λ*y*. *p2 y g*) (λ*y*. *p3 y g*) *f x*)
  ⟨*proof*⟩

fun
  xml3to4elements ::
    string ⇒ *'a xmlt* ⇒ *'b xmlt* ⇒ *'c xmlt* ⇒ *'d xmlt* ⇒ (*'a* ⇒ *'b* ⇒ *'c option* ⇒

```
'd ⇒ 'e) ⇒
    'e xmlt
where
  xml3to4elements tag p1 p2 p3 p4 f (XML name atts cs) = (
    let l = length cs in
    (if name = tag ∧ atts = [] ∧ l ≥ 3 ∧ l ≤ 4 then do {
      x1 ← p1 (cs ! 0);
      x2 ← p2 (cs ! 1);
      (if l = 4 then do {
        x3 ← p3 (cs ! 2);
        x4 ← p4 (cs ! 3);
        return (f x1 x2 (Some x3) x4)
      } else do {
        x4 ← p4 (cs ! 2);
        return (f x1 x2 None x4)
      } )
    } else Xmlt.fail tag (XML name atts cs))) |
  xml3to4elements tag p1 p2 p3 p4 f xml = Xmlt.fail tag xml
```

lemma *xml3to4elements-mono* [*partial-function-mono*]:
assumes *p1*: $\bigwedge$*y. mono-sum-bot (p1 y)*
$\bigwedge$*y. mono-sum-bot (p2 y)*
$\bigwedge$*y. mono-sum-bot (p3 y)*
$\bigwedge$*y. mono-sum-bot (p4 y)*
shows *mono-sum-bot (λg. xml3to4elements t (λy. p1 y g) (λy. p2 y g) (λy. p3 y g) (λy. p4 y g) f x)*
⟨*proof*⟩

```
fun many :: tag ⇒ 'a xmlt ⇒ ('a list ⇒ 'b) ⇒ 'b xmlt
where
  many tag p f (XML name atts cs) =
    (if name = tag ∧ atts = [] then (Xmlt.map p cs ≫= (return ∘ f))
    else Xmlt.fail tag (XML name atts cs)) |
  many tag p f xml = Xmlt.fail tag xml
hide-const (open) many
```

lemma *many-mono* [*partial-function-mono*]:
fixes *p1 :: xml ⇒ ('b ⇒ (string +⊥ 'c)) ⇒ string +⊥ 'd*
assumes $\bigwedge$*y. y ∈ set (Xml.children x) ⟹ mono-sum-bot (p1 y)*
shows *mono-sum-bot (λg. Xmlt.many t (λy. p1 y g) f x)*
⟨*proof*⟩

```
fun many1-gen :: tag ⇒ 'a xmlt ⇒ ('a ⇒ 'b xmlt) ⇒ ('a ⇒ 'b list ⇒ 'c) ⇒ 'c xmlt
where
  many1-gen tag p1 p2 f (XML name atts cs) =
    (if name = tag ∧ atts = [] ∧ cs ≠ [] then
      (case cs of h # t ⇒ do {
        x ← p1 h;
        xs ← Xmlt.map (p2 x) t;
```

```
      return (f x xs)
    })
  else Xmlt.fail tag (XML name atts cs)) |
many1-gen tag p1 p2 f xml = Xmlt.fail tag xml
```

definition *many1* :: *string* ⇒ *'a xmlt* ⇒ *'b xmlt* ⇒ (*'a* ⇒ *'b list* ⇒ *'c*) ⇒ *'c xmlt*
where
  many1 tag p1 p2 = *Xmlt.many1-gen tag p1* (λ-. *p2*)
hide-const (**open**) *many1*

lemma *many1-mono* [*partial-function-mono*]:
  fixes *p1* :: *xml* ⇒ (*'b* ⇒ (*string* $+_\perp$ *'c*)) ⇒ *string* $+_\perp$ *'d*
  assumes ⋀*y*. *mono-sum-bot* (*p1 y*)
    and ⋀*y*. *y* ∈ *set* (*tl* (*Xml.children x*)) ⟹ *mono-sum-bot* (*p2 y*)
  shows *mono-sum-bot* (λ*g*. *Xmlt.many1 t* (λ*y*. *p1 y g*) (λ*y*. *p2 y g*) *f x*)
  ⟨*proof*⟩

fun *length-ge-2* :: *'a list* ⇒ *bool*
where
  length-ge-2 (- # - # -) = *True* |
  length-ge-2 - = *False*

fun *many2* :: *tag* ⇒ *'a xmlt* ⇒ *'b xmlt* ⇒ *'c xmlt* ⇒ (*'a* ⇒ *'b* ⇒ *'c list* ⇒ *'d*) ⇒ *'d xmlt*
where
  many2 tag p1 p2 p3 f (*XML name atts cs*) =
    (*if name* = *tag* ∧ *atts* = [] ∧ *length-ge-2 cs then*
      (*case cs of cs0* # *cs1* # *t* ⇒ *do* {
        x ← *p1 cs0*;
        y ← *p2 cs1*;
        xs ← *Xmlt.map p3 t*;
        return (*f x y xs*)
      })
    else Xmlt.fail tag (*XML name atts cs*)) |
  many2 tag p1 p2 p3 f xml = *Xmlt.fail tag xml*

lemma *many2-mono* [*partial-function-mono*]:
  fixes *p1* :: *xml* ⇒ (*'b* ⇒ (*string* $+_\perp$ *'c*)) ⇒ *string* $+_\perp$ *'d*
  assumes ⋀*y*. *mono-sum-bot* (*p1 y*)
    and ⋀*y*. *mono-sum-bot* (*p2 y*)
    and ⋀*y*. *mono-sum-bot* (*p3 y*)
  shows *mono-sum-bot* (λ*g*. *Xmlt.many2 t* (λ*y*. *p1 y g*) (λ*y*. *p2 y g*) (λ*y*. *p3 y g*) *f x*)
  ⟨*proof*⟩

fun
  xml1or2many-elements ::

```
  string ⇒ 'a xmlt ⇒ 'b xmlt ⇒ 'c xmlt ⇒ ('a ⇒ 'b option ⇒ 'c list ⇒ 'd) ⇒ 'd
xmlt
where
 xml1or2many-elements tag p1 p2 p3 f (XML name atts cs) =
  (if name = tag ∧ atts = [] ∧ cs ≠ [] then
   (case cs of
    cs0 # tt ⇒
    do {
     x ← p1 cs0;
     (case tt of
      cs1 # t ⇒
      do {
       try do {
        y ← p2 cs1;
        xs ← Xmlt.map p3 t;
        return (f x (Some y) xs)
       } catch (λ -. do {
        xs ← Xmlt.map p3 tt;
        return (f x None xs)
       })
      }
     | [] ⇒ return (f x None []))})
   else Xmlt.fail tag (XML name atts cs)) |
 xml1or2many-elements tag p1 p2 p3 f  xml = Xmlt.fail tag xml

fun
 xml1many2elements-gen ::
  string ⇒ 'a xmlt ⇒ ('a ⇒ 'b xmlt) ⇒ 'c xmlt ⇒ 'd xmlt ⇒
   ('a ⇒ 'b list ⇒ 'c ⇒ 'd ⇒ 'e) ⇒ 'e xmlt
where
 xml1many2elements-gen tag p1 p2 p3 p4 f (XML name atts cs) = (
   let ds = List.rev cs; l = length cs in
   (if name = tag ∧ atts = [] ∧ l ≥ 3 then do {
    x ← p1 (cs ! 0);
    xs ← Xmlt.map (p2 x) (tl (take (l − 2) cs));
    y ← p3 (ds ! 1);
    z ← p4 (ds ! 0);
    return (f x xs y z)
   } else Xmlt.fail tag (XML name atts cs))) |
 xml1many2elements-gen tag p1 p2 p3 p4 f xml = Xmlt.fail tag xml

lemma xml1many2elements-gen-mono [partial-function-mono]:
 fixes p1 :: xml ⇒ ('b ⇒ (string +⊥ 'c)) ⇒ string +⊥ 'd
 assumes p1: ⋀y. mono-sum-bot (p1 y)
           ⋀y. mono-sum-bot (p3 y)
           ⋀y. mono-sum-bot (p4 y)
 shows mono-sum-bot (λg. xml1many2elements-gen t (λy. p1 y g) p2 (λy. p3 y
g) (λy. p4 y g) f x)
 ⟨proof⟩
```

fun
xml1many2elements ::
string ⇒ *'a xmlt* ⇒ *'b xmlt* ⇒ *'c xmlt* ⇒ *'d xmlt* ⇒ (*'a* ⇒ *'b list* ⇒ *'c* ⇒ *'d* ⇒ *'e*) ⇒
'e xmlt
where
xml1many2elements tag p1 p2 = *xml1many2elements-gen tag p1* (λ-. *p2*)

fun
xml-many2elements ::
string ⇒ *'a xmlt* ⇒ *'b xmlt* ⇒ *'c xmlt* ⇒ (*'a list* ⇒ *'b* ⇒ *'c* ⇒ *'d*) ⇒ *'d xmlt*
where
xml-many2elements tag p1 p2 p3 f (*XML name atts cs*) = (
let ds = *List.rev cs in*
(*if name* = *tag* ∧ *atts* = [] ∧ *length-ge-2 cs then do* {
xs ← *Xmlt.map p1* (*List.rev* (*tl* (*tl ds*)));
y ← *p2* (*ds* ! *1*);
z ← *p3* (*ds* ! *0*);
return (*f xs y z*)
} *else Xmlt.fail tag* (*XML name atts cs*))) |
xml-many2elements tag p1 p2 p3 f xml = *Xmlt.fail tag xml*

definition *options* :: (*string* × *'a xmlt*) *list* ⇒ *'a xmlt*
where
options ps x =
(*case map-of ps* (*Xml.tag x*) *of*
None ⇒ *error* (*concat*
[*''expected one of: ''*, *concat* (*map* (λ*p*. *fst p* @ *'' ''*) *ps*), *''*↩*''*, *''but found''*, *''*↩*''*, *show x*])
| *Some p* ⇒ *p x*)
hide-const (**open**) *options*

lemma *options-mono-gen* [*partial-function-mono*]:
assumes *p*: ⋀ *k p*. (*k*, *p*) ∈ *set ps* ⟹ *mono-sum-bot* (*p x*)
shows *mono-sum-bot* (λ *g*. *Xmlt.options* (*map* (λ (*k*, *p*). (*k*, (λ *y*. *p y g*))) *ps*) *x*)
⟨*proof*⟩

⟨*ML*⟩

declare *Xmlt.options-mono-thms* [*partial-function-mono*]

fun *choice* :: *string* ⇒ *'a xmlt list* ⇒ *'a xmlt*
where
choice e [] *x* = *error* (*concat* [*''error in parsing choice for ''*, *e*, *''*↩*''*, *show x*])
|
choice e (*p* # *ps*) *x* = (*try p x catch* (λ-. *choice e ps x*))
hide-const (**open**) *choice*

lemma *choice-mono-2* [*partial-function-mono*]:
assumes *p*: *mono-sum-bot* (*p1 x*)
mono-sum-bot (*p2 x*)
shows *mono-sum-bot* (λ *g. Xmlt.choice e* [(λ *y. p1 y g*), (λ *y. p2 y g*)] *x*)
⟨*proof*⟩

lemma *choice-mono-3* [*partial-function-mono*]:
assumes *p*: *mono-sum-bot* (*p1 x*)
mono-sum-bot (*p2 x*)
mono-sum-bot (*p3 x*)
shows *mono-sum-bot* (λ *g. Xmlt.choice e* [(λ *y. p1 y g*), (λ *y. p2 y g*), (λ *y. p3 y g*)] *x*)
⟨*proof*⟩

fun *change* :: *'a xmlt* $\Rightarrow$ (*'a* $\Rightarrow$ *'b*) $\Rightarrow$ *'b xmlt*
where
change p f x = *p x* $\gg=$ *return* $\circ$ *f*
hide-const (**open**) *change*

lemma *change-mono* [*partial-function-mono*]:
assumes *p*: $\bigwedge$*y. mono-sum-bot* (*p1 y*)
shows *mono-sum-bot* (λ*g. Xmlt.change* (λ*y. p1 y g*) *f x*)
⟨*proof*⟩

fun *int-of-digit* :: *char* $\Rightarrow$ *string* $+_\perp$ *int*
where
int-of-digit x =
(*if x* = *CHR ''0'' then return 0*
else if x = *CHR ''1'' then return 1*
else if x = *CHR ''2'' then return 2*
else if x = *CHR ''3'' then return 3*
else if x = *CHR ''4'' then return 4*
else if x = *CHR ''5'' then return 5*
else if x = *CHR ''6'' then return 6*
else if x = *CHR ''7'' then return 7*
else if x = *CHR ''8'' then return 8*
else if x = *CHR ''9'' then return 9*
else error (*x* # *'' is not a digit''*))

fun *int-of-string-aux* :: *int* $\Rightarrow$ *string* $\Rightarrow$ *string* $+_\perp$ *int*
where
int-of-string-aux n [] = *return n* |
int-of-string-aux n (*d* # *s*) = (*int-of-digit d* $\gg=$ (λ*m. int-of-string-aux* (*10* $*$ *n* + *m*) *s*))

definition *int-of-string* :: *string* $\Rightarrow$ *string* $+_\perp$ *int*
where
int-of-string s =

```
  (if s = [] then error ''cannot convert empty string into number''
  else if take 1 s = ''−'' then int-of-string-aux 0 (tl s) ≫= (λ i. return (0 − i))
  else int-of-string-aux 0 s)

hide-const int-of-string-aux

fun int :: tag ⇒ int xmlt
where
  int tag x = (Xmlt.text tag x ≫= int-of-string)
hide-const (open) int

fun nat :: tag ⇒ nat xmlt
where
  nat tag x = do {
    txt ← Xmlt.text tag x;
    i ← int-of-string txt;
    return (Int.nat i)
  }
hide-const (open) nat

definition rat :: rat xmlt
where
  rat = Xmlt.options [
    (''integer'', Xmlt.change (Xmlt.int ''integer'') of-int),
    (''rational'',
      Xmlt.pair ''rational'' (Xmlt.int ''numerator'') (Xmlt.int ''denominator'')
        (λ x y. of-int x / of-int y))]
hide-const (open) rat

end
```