Xml*

Christian Sternagel, René Thiemann and Akihisa Yamada

May 26, 2024

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* (λ*a*. '' '' +#+ *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*
unfolding *shows-attr-def* **by** (*cases av*) (*auto simp*: *show-law-simps*)

lemma *shows-attrs-append* [*show-law-simps*]:
shows-attrs as (*r* @ *s*) = *shows-attrs as r* @ *s*
using *shows-attr-append* **by** (*induct as*) (*simp-all add*: *shows-attrs-def*)

lemma *append-xml'*:
shows-XML-indent ind i xml (*r* @ *s*) = *shows-XML-indent ind i xml r* @ *s*
by (*induct xml arbitrary*: *ind r s*) (*auto simp*: *show-law-simps*)

lemma *shows-prec-xml-append* [*show-law-simps*]:
shows-prec d (*xml*::*xml*) (*r* @ *s*) = *shows-prec d xml r* @ *s*
unfolding *shows-prec-xml-def* **by** (*rule append-xml'*)

instance
by *standard* (*simp-all add*: *show-law-simps shows-list-xml-def*)

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*
by (*cases x*) (*auto simp*: *shows-prec-xmldoc-def show-law-simps*)

instance
by *standard* (*simp-all add*: *show-law-simps shows-list-xmldoc-def*)

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
  by (auto simp: parse-text-def)

lemma parse-text-consumes:
  assumes *: ts ≠ [] hd ts ≠ CHR ''<''
    and parse: parse-text ts = Inr (t, ts')
  shows length ts' < length ts
proof −
  from * obtain a tss where ts: ts = a # tss and not: a ≠ CHR ''<''
    by (cases ts, auto)
  note parse = parse [unfolded parse-text-def Let-def ts]
  from parse obtain x1 x2 where many: many ((≠) CHR ''<'') tss = Inr (x1,
x2)
    using not by (cases many ((≠) CHR ''<'') tss,
      auto simp: bind-def)
  from is-parser-many many have len: length x2 ≤ length tss by blast
  from parse many have length ts' ≤ length x2
    using not by (simp add: bind-def return-def split: if-splits)
  with len show ?thesis unfolding ts by auto
qed

definition parse-attribute-value :: string parser
where
  parse-attribute-value = do {
    exactly [CHR ''''];
    v ← many ((≠) CHR '''');
    exactly [CHR ''''];
    return v
  }

lemma is-parser-parse-attribute-value [intro]:
  is-parser parse-attribute-value
  by (auto simp: parse-attribute-value-def)
```

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-letter c ⟷ c ∈ set letters

lemma is-letter-pre-code:
  is-letter c ⟷
    CHR ''a'' ≤ c ∧ c ≤ CHR ''z'' ∨
```

```
    CHR ''A'' ≤ c ∧ c ≤ CHR ''Z'' ∨
    CHR ''0'' ≤ c ∧ c ≤ CHR ''9'' ∨
    c ∈ set ''-&;:−''
  by (cases c) (simp add: less-eq-char-def is-letter-def letters-def)

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

lemma many-letters [code, code-unfold]:
  many-letters = many is-letter
  by (simp add: is-letter-def [abs-def] manyof-def)

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
  fix s r x
  assume res: parse-name s = Inr (x, r)
  let ?exp = do {
    n ← many-letters;
    spaces;
    if n = [] then
      error (''expected letter '' @ letters @ '' but first symbol is '' @ take 1 s @ '''')
    else return n
  }
  have isp: is-parser ?exp by auto
  have id: parse-name s = ?exp s by (simp add: parse-name-def)
  from isp [unfolded is-parser-def, rule-format, OF res [unfolded id]]
  show length r ≤ length s .
qed

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))
 by pat-completeness auto

termination parse-attributes
proof
 show wf (measure length) by simp
next
 fix c s y ts ya tsa yb tsb
 assume pn: parse-name (c # s) = Inr (y, ts)
  and oo: exactly ''='' ts = Inr (ya, tsa)
  and pav: parse-attribute-value tsa = Inr (yb, tsb)
 have cp: is-cparser (exactly ''='') by auto
 from cp [unfolded is-cparser-def] oo have 1: length ts > length tsa by auto
 from is-parser-parse-name [unfolded is-parser-def] pn
  have 2: length (c # s) ≥ length ts by force
 from is-parser-parse-attribute-value [unfolded is-parser-def] pav
  have 3: length tsa ≥ length tsb by force
 from 1 2 3
  show (tsb, c # s) ∈ measure length
  by auto
qed

lemma is-parser-parse-attributes [intro]:
 is-parser parse-attributes
proof
 fix s r x
 assume parse-attributes s = Inr (x, r)
 then show length r ≤ length s
 proof (induct arbitrary: x rule: parse-attributes.induct)
  case (2 c s)
  show ?case
  proof (cases c ∈ set ''/>'')
   case True
   with 2(2) show ?thesis by simp
  next
   case False
   from False 2(2) obtain y1 s1
    where pn: parse-name (c # s) = Inr (y1, s1)
    by (cases parse-name (c # s)) (auto simp: bind-def)
   from False 2(2) pn obtain y2 s2
    where oo: exactly ''='' s1 = Inr (y2, s2)
    by (cases exactly ''='' s1) (auto simp: bind-def)
   from False 2(2) pn oo obtain y3 s3
    where pav: parse-attribute-value s2 = Inr (y3, s3)
    by (cases parse-attribute-value s2) (auto simp: bind-def)
   from False 2(2) pn oo pav obtain y4
```

```
      where patts: parse-attributes s3 = Inr (y4, r)
      by (cases parse-attributes s3) (auto simp: return-def bind-def)
    have length r ≤ length s3 using 2(1)[OF False pn oo pav patts] .
    also have ... ≤ length s2
      using is-parser-parse-attribute-value [unfolded is-parser-def] pav by auto
    also have ... ≤ length s1 using is-parser-exactly [unfolded is-parser-def] oo
by auto
    also have ... ≤ length (c # s)
      using is-parser-parse-name [unfolded is-parser-def] pn by force
    finally show length r ≤ length (c # s) by auto
  qed
 qed simp
qed

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)
 by pat-completeness auto

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 (induct rule: wf-induct [where P = ?prop and r = measure length])
  fix s
  assume ∀ t. (t, s) ∈ measure length ⟶ ?prop t
  then have ind1: ⋀ t. length t < length s ⟹ parse-nodes-dom t
    and ind2: ⋀ t x r. length t < length s ⟹ parse-nodes t = Inr (x,r) ⟹ length
r ≤ length t by auto
  let ?check = λ s. s = [] ∨ take 2 s = ''</''
  let ?check2 = hd s ≠ CHR ''<''
  have dom: parse-nodes-dom s
  proof
    fix y
    assume parse-nodes-rel y s
    then show parse-nodes-dom y
    proof
      fix ts ya tsa
      assume ∗: y = tsa  s = ts  ¬ (ts = [] ∨ take 2 ts = ''</'')
        hd ts ≠ CHR ''<'' and parse: parse-text ts = Inr (ya, tsa)
      from parse-text-consumes[OF - - parse] ∗(3−4) have length tsa < length ts
by auto
      with ∗ have len: length s > length y by simp
      from ind1[OF this] show parse-nodes-dom y .
    next
      fix ts ya tsa yaa tsb yb tsc yc tsd
      assume y = tsd and s = ts and ¬ ?check ts
        and exactly ''<'' ts = Inr (ya, tsa)
        and parse-name tsa = Inr (yaa, tsb)
        and parse-attributes tsb = Inr (yb, tsc)
        and oneof [''/>'', ''>''] tsc = Inr (yc, tsd)
        and yc = ''/>''
      then have len: length s > length y
        using is-cparser-exactly [of ''<'']
        and is-parser-oneof [of [''/>'', ''>'']]
        and is-parser-parse-attributes
        and is-parser-parse-name
        by (auto dest!: is-parser-length is-cparser-length)
      with ind1[OF len] show parse-nodes-dom y by simp
    next
      fix ts ya tsa yaa tsb yb tsc yc tsd
      assume y = tsd and s = ts and ¬ ?check ts
        and exactly ''<'' ts = Inr (ya, tsa)
        and parse-name tsa = Inr (yaa, tsb)
        and parse-attributes tsb = Inr (yb, tsc)
        and oneof [''/>'', ''>''] tsc = Inr (yc, tsd)
      then have len: length s > length y
        using is-cparser-exactly [of ''<'', simplified]
        and is-parser-oneof [of [''/>'', ''>'']]
        and is-parser-parse-attributes
```

```
      and is-parser-parse-name
      by (auto dest!: is-parser-length is-cparser-length)
    with ind1[OF len] show parse-nodes-dom y by simp
  next
    fix ts ya tsa yaa tsb yb tsc yc tse ye tsf yf tsg yg tsh yh tsi yi tsj
    assume y: y = tsj and s = ts and ¬ ?check ts
      and exactly ''<'' ts = Inr (ya, tsa)
      and parse-name tsa = Inr (yaa, tsb)
      and parse-attributes tsb = Inr (yb, tsc)
      and oneof [''/>'', ''>''] tsc = Inr (yc, tse)
      and rec: parse-nodes-sumC tse = Inr (ye, tsf)
      and last: exactly ''</'' tsf = Inr (yf, tsg)
        exactly yaa tsg = Inr (yg, tsh)
        exactly ''>'' tsh = Inr (yi, tsj)
    then have len: length s > length tse
      using is-cparser-exactly [of ''<'', simplified]
      and is-parser-oneof [of [''/>'', ''>'']]
      and is-parser-parse-attributes
      and is-parser-parse-name
      by (auto dest!: is-parser-length is-cparser-length)
    from last(1) last(2) have len2a: length tsf ≥ length tsh
      using is-parser-exactly [of ''</''] and is-parser-exactly [of yaa]
      and is-parser-parse-name by (auto dest!: is-parser-length)
    have len2c: length tsh ≥ length y using last(3)
      using is-parser-exactly [of ''>''] by (auto simp: y dest!: is-parser-length)
    from len2a len2c have len2: length tsf ≥ length y by simp
     from ind2[OF len rec[unfolded parse-nodes-def[symmetric]]] len len2 have
length s > length y by simp
    from ind1[OF this]
    show parse-nodes-dom y .
  qed
 qed
 note psimps = parse-nodes.psimps[OF dom]
 show ?prop s
 proof (intro conjI, rule dom, intro allI impI)
  fix x r
  assume res: parse-nodes s = Inr (x,r)
  note res = res[unfolded psimps]
  then show length r ≤ length s
  proof (cases ?check s)
    case True
    then show ?thesis using res by (simp add: return-def)
  next
    case False note oFalse = this
    show ?thesis
    proof (cases ?check2)
      case True
      note res = res[simplified False True, simplified]
       from res obtain y1 s1 where pt: parse-text s = Inr (y1, s1) by (cases
```

parse-text s, *auto simp*: *bind-def*)
note *res* = *res*[*unfolded bind-def pt*, *simplified*]
from *res* **obtain** *y2 s2*
where *pn*: *parse-nodes s1* = *Inr* (*y2*, *s2*)
by (*cases parse-nodes s1*) (*auto simp*: *bind-def*)
note *res* = *res*[*simplified bind-def pn*, *simplified*]
from *res* **have** *r*: *r* = *s2* **by** (*simp add*: *return-def bind-def*)
from *parse-text-consumes*[*OF - True pt*] *False*
have *lens*: *length s1* < *length s* **by** *auto*
from *ind2*[*OF lens pn*] **have** *length s2* ≤ *length s1* **.**
then show *?thesis* **using** *lens* **unfolding** *r* **by** *auto*
next
case *False* **note** *ooFalse* = *this*
note *res* = *res*[*simplified oFalse ooFalse*, *simplified*]
from *res* **obtain** *y1 s1* **where** *oo*: *exactly* ''<'' *s* = *Inr* (*y1*, *s1*) **by** (*cases exactly* ''<'' *s*, *auto simp*: *bind-def*)
note *res* = *res*[*unfolded bind-def oo*, *simplified*]
from *res* **obtain** *y2 s2*
where *pn*: *parse-name s1* = *Inr* (*y2*, *s2*)
by (*cases parse-name s1*) (*auto simp*: *bind-def psimps*)
note *res* = *res*[*simplified bind-def pn*, *simplified*]
from *res* **obtain** *y3 s3* **where** *pa*: *parse-attributes s2* = *Inr* (*y3*, *s3*)
by (*cases parse-attributes s2*) (*auto simp*: *return-def bind-def*)
note *res* = *res*[*simplified pa*, *simplified*]
from *res* **obtain** *y4 s4*
where *oo2*: *oneof* [''/>'', ''>''] *s3* = *Inr* (*y4*, *s4*)
by (*cases oneof* [''/>'', ''>''] *s3*) (*auto simp*: *return-def bind-def*)
note *res* = *res*[*unfolded oo2*, *simplified*]
from *is-parser-parse-attributes* **and** *is-parser-oneof* [*of* [''/>'', ''>'']]
and *is-cparser-exactly* [*of* ''<'', *simplified*] **and** *is-parser-parse-name*
and *oo pn pa oo2*
have *s-s4*: *length s* > *length s4*
by (*auto dest*!: *is-parser-length is-cparser-length*)
show *?thesis*
proof (*cases y4* = ''/>'')
case *True*
from *res True* **obtain** *y5*
where *pns*: *parse-nodes s4* = *Inr* (*y5*, *r*)
by (*cases parse-nodes s4*) (*auto simp*: *return-def bind-def*)
from *ind2*[*OF s-s4 pns*] *s-s4* **show** *length r* ≤ *length s* **by** *simp*
next
case *False*
note *res* = *res*[*simplified False*, *simplified*]
from *res* **obtain** *y6 s6* **where** *pns*: *parse-nodes s4* = *Inr* (*y6*, *s6*)
by (*cases parse-nodes s4*) (*auto simp*: *return-def bind-def*)
note *res* = *res*[*unfolded bind-def pns*, *simplified*, *unfolded bind-def*]
from *res* **obtain** *y7 s7* **where** *oo3*: *exactly* ''</'' *s6* = *Inr* (*y7*, *s7*) **by** (*cases exactly* ''</'' *s6*, *auto*)
note *res* = *res*[*unfolded oo3*, *simplified*, *unfolded bind-def*,

```
        simplified, unfolded bind-def]
      from res obtain y8 s8 where oo4: exactly y2 s7 = Inr (y8, s8) by (cases
exactly y2 s7, auto)
       note res = res[unfolded oo4 bind-def, simplified]
       from res obtain y10 s10 where oo5: exactly ''>'' s8 = Inr (y10,s10)
        by (cases exactly ''>'' s8, auto simp: bind-def)
       note res = res[unfolded oo5 bind-def, simplified]
       from res obtain y11 s11 where pns2: parse-nodes s10 = Inr (y11, s11)
by (cases parse-nodes s10, auto simp: bind-def)
       note res = res[unfolded bind-def pns2, simplified]
       note one = is-parser-oneof [unfolded is-parser-def, rule-format]
       note exact = is-parser-exactly [unfolded is-parser-def, rule-format]
       from ind2[OF s-s4 pns] s-s4 exact[OF oo3] exact[OF oo4]
        have s-s7: length s > length s8 unfolding is-parser-def by force
       with exact[OF oo5] have s-s10: length s > length s10 by simp
       with ind2[OF s-s10 pns2] have s-s11: length s > length s11 by simp
       then show length r ≤ length s using res by (auto simp: return-def)
      qed
     qed
   qed
  qed
qed simp

termination parse-nodes using parse-nodes-help by blast

lemma parse-nodes [intro]:
  is-parser parse-nodes
  unfolding is-parser-def using parse-nodes-help by blast
```

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 (rule ext)
  fix xs
  have id: ''one of '' @ shows-list [''/>'', ''>''] [] = ''one of [/>, >]''
   by (simp add: shows-list-list-def showsp-list-def pshowsp-list-def shows-list-gen-def
                 shows-string-def shows-prec-list-def shows-list-char-def)
  note d = oneof-def oneof-aux.simps id
  show ?l xs = ?r xs
  proof (cases xs)
```

```
    case Nil
    show ?thesis unfolding Nil d by simp
  next
    case (Cons x xs) note oCons = this
    show ?thesis
    proof (cases x = CHR ''>'')
      case True
      show ?thesis unfolding Cons d True by simp
    next
      case False note oFalse = this
      show ?thesis
      proof (cases x = CHR ''/'')
        case False
        show ?thesis unfolding Cons d using False oFalse by simp
      next
        case True
        show ?thesis
        proof (cases xs)
          case Nil
          show ?thesis unfolding Cons Nil d by auto
        next
          case (Cons y ys)
          show ?thesis unfolding oCons Cons d by simp
        qed
      qed
    qed
  qed
qed

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)
  by (intro ext) auto

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)
  by pat-completeness auto

termination parse-header
proof
  fix ts y tsa
  assume scan-upto ''?>'' ts = Inr (y, tsa)
  with is-cparser-scan-upto have length ts > length tsa
    unfolding is-cparser-def by force
  then show (tsa, ts) ∈ measure length by simp
qed simp


definition comment-error (x :: unit) = Code.abort (STR ''comment not terminated'') (λ -. Nil :: string)
definition comment-error-hyphen (x :: unit) = Code.abort (STR ''double hyphen within comment'') (λ -. Nil :: string)

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*
unfolding *remove-comments-def rc-open-1-def* **..**

lemma *char-eq-via-integer-eq*: *c* = *d* ⟷ *integer-of-char c* = *integer-of-char d*
unfolding *integer-of-char-def* **by** *simp*

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*
by *code-simp+*

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* ()
unfolding
rc-open-1-def

```
    rc-open-2-def
    rc-open-3-def
    rc-open-4-def
    rc-close-1-def
    rc-close-2-def
    rc-close-3-def
  by (simp-all add: char-eq-via-integer-eq Let-def)


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)
  unfolding trim-def
  apply (rule arg-cong[of - - dropWhile], rule ext)
  unfolding Let-def in-set-simps less-eq-char-code char-eq-via-integer-eq
  by (auto simp: integer-of-char-def Let-def)

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)))
  by (induct xs arbitrary: ys, auto)
```

lemma *many-take-drop*: *many f xs* = *Inr* (*takeWhile f xs*, *dropWhile f xs*)
by (*induct f xs rule*: *many.induct*, *auto*)

lemma *trim-takeWhile-inside*: *trim* (*takeWhile* ((≠) *CHR* ''<'') *cs*) = *takeWhile* ((≠) *CHR* ''<'') (*trim cs*)
unfolding *trim-def* **by** (*induct cs*, *auto*)

lemma *trim-dropWhile-inside*: *dropWhile* ((≠) *CHR* ''<'') *cs* = *dropWhile* ((≠) *CHR* ''<'') (*trim cs*)
unfolding *trim-def* **by** (*induct cs*, *auto*)

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

lemma *parse-text-code*[*code*]: *parse-text cs* = *parse-text-impl cs*
proof −
define *xs* **where** *xs* = *trim cs*
show *?thesis*
unfolding *parse-text-def*
unfolding *Parser-Monad.bind-def Error-Monad.bind-def*
unfolding *Let-def*
unfolding *many-take-drop sum.simps split*
unfolding *trim-takeWhile-inside trim-dropWhile-inside*[*of cs*] *Parser-Monad.return-def*
unfolding *parse-text-impl-def*
unfolding *xs-def*[*symmetric*]
unfolding *parse-text-main split*
apply (*simp*, *intro conjI impI*, *force simp*: *trim-def*)
proof
define *ys* **where** *ys* = *takeWhile* ((≠) *CHR* ''<'') *xs*
assume *trim* (*rev* (*takeWhile* ((≠) *CHR* ''<'') *xs*)) = []
and *takeWhile* ((≠) *CHR* ''<'') *xs* ≠ []
hence *trim* (*rev ys*) = [] **and** *ys* ≠ [] **unfolding** *ys-def* **by** *auto*
from *this*(*1*) **have** *ys*: ⋀ *y*. *y* ∈ *set ys* ⟹ *y* ∈ *set wspace* **unfolding** *trim-def*
by *simp*
with ‹*ys* ≠ []› **show** *False* **unfolding** *ys-def xs-def trim-def*
by (*metis* (*no-types*, *lifting*) *dropWhile-eq-Nil-conv dropWhile-idem trim-def trim-takeWhile-inside xs-def*)
qed
qed

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*))
unfolding *parse-text-main.simps* **by** (*auto simp*: *char-eq-via-integer-eq*)

lemma *exactly-head*: *exactly* [*c*] (*c* # *cs*) = *Inr* ([*c*],*trim cs*)
unfolding *exactly-def* **by** *simp*

lemma *take-1-test*: (*case cs of* [] ⇒ *False* | *c* # *x* ⇒ *c* = *CHR* ''/'') = (*take 1 cs* = ''/'')
by (*cases cs*, *auto*)

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*))
unfolding *exactly-close-def exactly-def exactly-aux.simps* **by** (*auto simp*: *char-eq-via-integer-eq*)

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*))
unfolding *exactly-end-def exactly-def exactly-aux.simps* **by** (*auto simp*: *char-eq-via-integer-eq*)

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 (*intro ext*)
fix *xs*
show *?l xs* = *?r xs* **unfolding** *Parser-Monad.bind-def Error-Monad.bind-def*
by (*cases xs*, *auto split*: *sum.splits simp*: *err-expecting-def*)
qed

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))))
  unfolding oneof-closed-combined.simps Let-def
  by (auto split: list.splits simp: char-eq-via-integer-eq)

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 ≫=
                (λn. parse-attributes ≫=
                    (λatts.
                                    oneof-closed-combined (parse-nodes ≫= (λcs.
Parser-Monad.return (XML n atts [] # cs)))
                                (parse-nodes ≫=
                                    (λcs. exactly-end ≫=
                                        (λ-. exactly n ≫=
                                            (λ-. exactly-close ≫=
                                                    (λ-. parse-nodes ≫= (λns.
Parser-Monad.return (XML n atts cs # ns)))))))))))
            (trim cs)
    else (parse-text ≫= (λt. parse-nodes ≫= (λns. Parser-Monad.return (XML-text
(the t) # ns)))) (c # cs))
  unfolding parse-nodes-current-code[of c # cs] exactly-close-def exactly-end-def
oneof-closed-combined
  by (simp-all add: Parser-Monad.bind-def exactly-head take-1-test)

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 ≫=
                (λn. parse-attributes ≫=
                    (λatts.
                                    oneof-closed-combined (parse-nodes ≫= (λcs.
Parser-Monad.return (XML n atts [] # cs)))
                                (parse-nodes ≫=
                                    (λcs. exactly-end ≫=
                                        (λ-. exactly n ≫=
```

```
                                        (λ-. exactly-close ≫=
                                                (λ-. parse-nodes ≫= (λns.
Parser-Monad.return (XML n atts cs # ns))))))))))
            (trim cs)
    else (parse-text ≫= (λt. parse-nodes ≫= (λns. Parser-Monad.return (XML-text
(the t) # ns)))) (c # cs))
  unfolding parse-nodes-pre-code
  unfolding Let-def by (auto simp: char-eq-via-integer-eq)

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 ≫=
       (λk. exactly ''='' ≫= (λ-. parse-attribute-value ≫= (λv. parse-attributes ≫=
(λatts. Parser-Monad.return ((k, v) # atts))))))
       (c # s)))
  unfolding parse-attributes.simps
  unfolding Let-def in-set-simps
  by (auto simp: char-eq-via-integer-eq)

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 −
  define d where d = integer-of-char c
  have d ≤ 59 ⟷ (d ≤ 57 ∨ d = 58 ∨ d = 59) for d :: int by auto
  hence d ≤ 59 ⟷ (d ≤ 57 ∨ d = 58 ∨ d = 59)
   by (metis int-of-integer-numeral integer-eqI integer-less-eq-iff verit-comp-simplify1(2))
  thus ?thesis
    unfolding is-letter-pre-code in-set-simps Let-def d-def
      less-eq-char-code char-eq-via-integer-eq
    unfolding integer-of-char-def
    by auto
qed


declare spaces-def[code-unfold del]

lemma spaces-code[code]:
  spaces cs = Inr ((), trim cs)
 unfolding spaces-def trim-def manyof-def many-take-drop Parser-Monad.bind-def
Parser-Monad.return-def by auto
```

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*)
  unfolding *many-letters-def manyof-def many-take-drop*
  by (*rule arg-cong*[*of - - Inr*], *rule sym*, *induct cs*, *auto simp*: *is-letter-def*)

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*))
  unfolding *parse-name-def many-letters-code spaces-code*
    Parser-Monad.bind-def Error-Monad.bind-def sum.simps split
    Parser-Monad.error-def Parser-Monad.return-def if-distribR **by** *auto*

end

theory *Xmlt*
  imports
    HOL−Library.Extended-Nat
    Show.Number-Parser
    Certification-Monads.Strict-Sum
    Show.Shows-Literal
    Xml
begin

String literals in parser, for nicer generated code

type-synonym *ltag* = *String.literal*

datatype *'a xml-error* = *TagMismatch ltag list* | *Fatal 'a*

TagMismatch tags represents tag mismatch, expecting one of *tags* but something else is encountered.

lemma *xml-error-mono* [*partial-function-mono*]:
  assumes *p1*: ⋀*tags*. *mono-option* (*p1 tags*)
    and *p2*: ⋀*x*. *mono-option* (*p2 x*)
    and *f*: *mono-option f*
  shows *mono-option* (λ*g*. *case s of TagMismatch tags* ⇒ *p1 tags g* | *Fatal x* ⇒ *p2 x g*)
  using *assms* **by** (*cases s*, *auto intro*!:*partial-function-mono*)

A state is a tuple of the XML or list of XMLs to be parsed, the attributes,

a flag indicating if mismatch is allowed, a list of tags that have been mismatched, the current position.

type-synonym *'a xmlt = xml × (string × string) list × bool × ltag list × ltag list ⇒ String.literal xml-error +⊥ 'a*
type-synonym *'a xmlst = xml list × (string × string) list × bool × ltag list × ltag list ⇒ String.literal xml-error +⊥ 'a*

lemma *xml-state-cases*:
  assumes ⋀*p nam atts xmls. x = (XML nam atts xmls, p) ⟹ thesis*
    and ⋀*p txt. x = (XML-text txt, p) ⟹ thesis*
  shows *thesis*
  using *assms* **by** (*cases x*; *cases fst x*, *auto*)

lemma *xmls-state-cases*:
  assumes ⋀*p. x = ([],p) ⟹ thesis*
    and ⋀*xml xmls p. x = (xml # xmls, p) ⟹ thesis*
  shows *thesis*
  using *assms* **by** (*cases x*; *cases fst x*, *auto*)

lemma *xmls-state-induct*:
  fixes *x :: xml list × -*
  assumes ⋀*a b c d. P ([],a,b,c,d)*
    and ⋀*xml xmls a b c d.* (⋀*a b c d. P (xmls,a,b,c,d)*) ⟹ *P (xml # xmls, a, b, c, d)*
  shows *P x*
proof (*induct x*)
  case (*fields xmls a b c d*)
  with *assms* **show** *?case* **by** (*induct xmls arbitrary:a b c d*, *auto*)
qed

definition *xml-error*
  where *xml-error str x ≡ case x of (xmls,-,-,-,pos) ⇒*
  let next = case xmls of
      XML tag - - # - ⇒ STR ''<'' + String.implode tag + STR ''>''
    | *XML-text str # - ⇒ STR ''text element '' + String.implode str + STR ''''*
    | *[] ⇒ STR ''tag close''*
  in
  Left (Fatal (STR ''parse error on '' + next + STR '' at '' + default-showsl-list showsl-lit pos (STR '''') + STR '':↩'' + str))

definition *xml-return :: 'a ⇒ 'a xmlst*
  where *xml-return v x ≡ case x*
  of ([],-) ⇒ Right v
  | - ⇒ xml-error (STR ''expecting tag close'') x

definition *mismatch tag x ≡ case x of*
  (xmls,atts,flag,cands,-) ⇒
    if flag then Left (TagMismatch (tag#cands))
     else xml-error (STR ''expecting '' + default-showsl-list showsl-lit (tag#cands)

(*STR* ''')) *x*

abbreviation *xml-any* :: *xml xmlt*
where
xml-any x ≡ *Right* (*fst x*)

Conditional parsing depending on tag match.

definition *bind2* :: *'a* +⊥ *'b* ⇒ (*'a* ⇒ *'c* +⊥ *'d*) ⇒ (*'b* ⇒ *'c* +⊥ *'d*) ⇒ *'c* +⊥ *'d*
where
bind2 x f g = (*case x of*
Bottom ⇒ *Bottom*
| *Left a* ⇒ *f a*
| *Right b* ⇒ *g b*)

lemma *bind2-cong*[*fundef-cong*]: *x* = *y* ⟹ (⋀ *a*. *y* = *Left a* ⟹ *f1 a* = *f2 a*) ⟹

(⋀ *b*. *y* = *Right b* ⟹ *g1 b* = *g2 b*) ⟹ *bind2 x f1 g1* = *bind2 y f2 g2*
by (*cases x*, *auto simp*: *bind2-def*)

lemma *bind2-code*[*code*]:
bind2 (*sumbot a*) *f g* = (*case a of Inl a* ⇒ *f a* | *Inr b* ⇒ *g b*)
by (*cases a*) (*auto simp*: *bind2-def*)

definition *xml-or* (**infixr** *XMLor 51*)
where
xml-or p1 p2 x ≡ *case x of* (*x1*,*atts*,*flag*,*cands*,*rest*) ⇒ (
bind2 (*p1* (*x1*,*atts*,*True*,*cands*,*rest*))
(λ *err1*. *case err1*
of TagMismatch cands1 ⇒ *p2* (*x1*,*atts*,*flag*,*cands1*,*rest*)
| *err1* ⇒ *Left err1*)
Right)

definition *xml-do* :: *ltag* ⇒ *'a xmlst* ⇒ *'a xmlt* **where**
xml-do tag p x ≡
case x of (*XML nam atts xmls*, -, *flag*, *cands*, *pos*) ⇒
if nam = *String.explode tag then p* (*xmls*,*atts*,*False*,[],*tag*#*pos*) — inner tag mismatch is not allowed
else mismatch tag ([*fst x*], *snd x*)
| - ⇒ *mismatch tag* ([*fst x*], *snd x*)

parses the first child

definition *xml-take* :: *'a xmlt* ⇒ (*'a* ⇒ *'b xmlst*) ⇒ *'b xmlst*
where *xml-take p1 p2 x* ≡
case x of ([],*rest*) ⇒ (
— Only for accumulating expected tags.
bind2 (*p1* (*XML* [] [] [], *rest*)) *Left* (λ *a*. *Left* (*Fatal* (*STR* ''*unexpected*''))))
)
| (*x*#*xs*,*atts*,*flag*,*cands*,*rest*) ⇒ (
bind2 (*p1* (*x*,*atts*,*flag*,*cands*,*rest*)) *Left*

(λ *a*. *p2 a* (*xs*,*atts*,*False*,[],*rest*))) — If one child is parsed, then later mismatch is not allowed

definition *xml-take-text* :: (*string* ⇒ *'a xmlst*) ⇒ *'a xmlst* **where**
xml-take-text p xs ≡
case xs of (*XML-text text* # *xmls*, *s*) ⇒ *p text* (*xmls*,*s*)
| - ⇒ *xml-error* (*STR ''expecting a text''*) *xs*

definition *xml-take-int* :: (*int* ⇒ *'a xmlst*) ⇒ *'a xmlst* **where**
xml-take-int p xs ≡
case xs of (*XML-text text* # *xmls*, *s*) ⇒
(*case int-of-string text of Inl x* ⇒ *xml-error x xs* | *Inr n* ⇒ *p n* (*xmls*,*s*))
| - ⇒ *xml-error* (*STR ''expecting an integer''*) *xs*

definition *xml-take-nat* :: (*nat* ⇒ *'a xmlst*) ⇒ *'a xmlst* **where**
xml-take-nat p xs ≡
case xs of (*XML-text text* # *xmls*, *s*) ⇒
(*case nat-of-string text of Inl x* ⇒ *xml-error x xs* | *Inr n* ⇒ *p n* (*xmls*,*s*))
| - ⇒ *xml-error* (*STR ''expecting a number''*) *xs*

definition *xml-leaf* **where**
xml-leaf tag ret ≡ *xml-do tag* (*xml-return ret*)

definition *xml-text* :: *ltag* ⇒ *string xmlt* **where**
xml-text tag ≡ *xml-do tag* (*xml-take-text xml-return*)

definition *xml-int* :: *ltag* ⇒ *int xmlt* **where**
xml-int tag ≡ *xml-do tag* (*xml-take-int xml-return*)

definition *xml-nat* :: *ltag* ⇒ *nat xmlt* **where**
xml-nat tag ≡ *xml-do tag* (*xml-take-nat xml-return*)

definition *bool-of-string* :: *string* ⇒ *String.literal* + *bool*
where
bool-of-string s ≡
if s = *''true''* *then Inr True*
else if s = *''false''* *then Inr False*
else Inl (*STR ''cannot convert ''* + *String.implode s* + *STR '' into Boolean''*)

definition *xml-bool* :: *ltag* ⇒ *bool xmlt*
where
xml-bool tag x ≡
bind2 (*xml-text tag x*) *Left*
(λ *str*. (*case bool-of-string str of Inr b* ⇒ *Right b*
| *Inl err* ⇒ *xml-error err* ([*fst x*], *snd x*)
))

definition *xml-change* :: *'a xmlt* ⇒ (*'a* ⇒ *'b xmlst*) ⇒ *'b xmlt* **where**

```
  xml-change p f x ≡
    bind2 (p x) Left (λ a. case x of (-,rest) ⇒ f a ([],rest))
```

Parses the first child, if tag matches.

```
definition xml-take-optional :: 'a xmlt ⇒ ('a option ⇒ 'b xmlst) ⇒ 'b xmlst
  where xml-take-optional p1 p2 xs ≡
  case xs of ([],-) ⇒ p2 None xs
  | (xml # xmls, atts, allow, cands, rest) ⇒
    bind2 (p1 (xml, atts, True, cands, rest))
      (λ e. case e of
            TagMismatch cands1 ⇒ p2 None (xml#xmls, atts, allow, cands1, rest)
— TagMismatch is allowed
          | - ⇒ Left e)
      (λ a. p2 (Some a) (xmls, atts, False, [], rest))

definition xml-take-default :: 'a ⇒ 'a xmlt ⇒ ('a ⇒ 'b xmlst) ⇒ 'b xmlst
  where xml-take-default a p1 p2 xs ≡
  case xs of ([],-) ⇒ p2 a xs
  | (xml # xmls, atts, allow, cands, rest) ⇒ (
    bind2 (p1 (xml, atts, True, cands, rest))
      (λ e. case e of
            TagMismatch cands1 ⇒ p2 a (xml#xmls, atts, allow, cands1, rest) —
TagMismatch is allowed
          | - ⇒ Left e)
      (λa. p2 a (xmls, atts, False, [], rest)))
```

Take first children, as many as tag matches.

```
fun xml-take-many-sub :: 'a list ⇒ nat ⇒ enat ⇒ 'a xmlt ⇒ ('a list ⇒ 'b xmlst)
⇒ 'b xmlst where
  xml-take-many-sub acc minOccurs maxOccurs p1 p2 ([], atts, allow, rest) = (
    if minOccurs = 0 then p2 (rev acc) ([], atts, allow, rest)
    else — only for nice error log
        bind2 (p1 (XML [] [] [], atts, False, rest)) Left (λ -. Left (Fatal (STR
''unexpected'')))
  )
| xml-take-many-sub acc minOccurs maxOccurs p1 p2 (xml # xmls, atts, allow,
cands, rest) = (
      if maxOccurs = 0 then p2 (rev acc) (xml # xmls, atts, allow, cands, rest)
      else
        bind2 (p1 (xml, atts, minOccurs = 0, cands, rest))
          (λ e. case e of
                TagMismatch tags ⇒ p2 (rev acc) (xml # xmls, atts, allow, cands,
rest)
              | - ⇒ Left e)
          (λ a. xml-take-many-sub (a # acc) (minOccurs−1) (maxOccurs−1) p1 p2
(xmls, atts, False, [], rest))
  )

abbreviation xml-take-many where xml-take-many ≡ xml-take-many-sub []
```

fun *pick-up* **where**
  pick-up rest key [] = *None*
| *pick-up rest key* ((*l*,*r*)#*s*) = (*if key* = *l then Some* (*r*,*rest*@*s*) *else pick-up* ((*l*,*r*)#*rest*) *key s*)

definition *xml-take-attribute* :: *ltag* ⇒ (*string* ⇒ *'a xmlst*) ⇒ *'a xmlst*
  where *xml-take-attribute att p xs* ≡
  case xs of (*xmls*,*atts*,*allow*,*cands*,*pos*) ⇒ (
    case pick-up [] (*String.explode att*) *atts of*
      None ⇒ *xml-error* (*STR ''attribute '' + att + STR '' not found''*) *xs*
    | *Some*(*v*,*rest*) ⇒ *p v* (*xmls*,*rest*,*allow*,*cands*,*pos*)
)

definition *xml-take-attribute-optional* :: *ltag* ⇒ (*string option* ⇒ *'a xmlst*) ⇒ *'a xmlst*
  where *xml-take-attribute-optional att p xs* ≡
  case xs of (*xmls*,*atts*,*info*) ⇒ (
    case pick-up [] (*String.explode att*) *atts of*
      None ⇒ *p None xs*
    | *Some*(*v*,*rest*) ⇒ *p* (*Some v*) (*xmls*,*rest*,*info*)
)

definition *xml-take-attribute-default* :: *string* ⇒ *ltag* ⇒ (*string* ⇒ *'a xmlst*) ⇒ *'a xmlst*
  where *xml-take-attribute-default def att p xs* ≡
  case xs of (*xmls*,*atts*,*info*) ⇒ (
    case pick-up [] (*String.explode att*) *atts of*
      None ⇒ *p def xs*
    | *Some*(*v*,*rest*) ⇒ *p v* (*xmls*,*rest*,*info*)
)

nonterminal *xml-binds* **and** *xml-bind*
syntax
  -xml-block :: *xml-binds* ⇒ *'a* (*XMLdo* {//(*2* -)//} [*12*] *1000*)
  -xml-take :: *pttrn* ⇒ *'a* ⇒ *xml-bind* ((*2*- ←/ -) *13*)
  -xml-take-text :: *pttrn* ⇒ *xml-bind* ((*2*- ←*text*) *13*)
  -xml-take-int :: *pttrn* ⇒ *xml-bind* ((*2*- ←*int*) *13*)
  -xml-take-nat :: *pttrn* ⇒ *xml-bind* ((*2*- ←*nat*) *13*)
  -xml-take-att :: *pttrn* ⇒ *ltag* ⇒ *xml-bind* ((*2*- ←*att*/ -) *13*)
  -xml-take-att-optional :: *pttrn* ⇒ *ltag* ⇒ *xml-bind* ((*2*- ←*att*?/ -) *13*)
  -xml-take-att-default :: *pttrn* ⇒ *ltag* ⇒ *string* ⇒ *xml-bind* ((*2*- ←*att*[(-)]/ -) *13*)
  -xml-take-optional :: *pttrn* ⇒ *'a* ⇒ *xml-bind* ((*2*- ←?/ -) *13*)
  -xml-take-default :: *pttrn* ⇒ *'b* ⇒ *'a* ⇒ *xml-bind* ((*2*- ←[(-)]/ -) *13*)
  -xml-take-all :: *pttrn* ⇒ *'a* ⇒ *xml-bind* ((*2*- ←*/ -) *13*)
  -xml-take-many :: *pttrn* ⇒ *nat* ⇒ *enat* ⇒ *'a* ⇒ *xml-bind* ((*2*- ←^{(-)..(-)}/ -) *13*)
  -xml-let :: *pttrn* ⇒ *'a* ⇒ *xml-bind* ((*2let* - =/ -) [*1000*, *13*] *13*)
  -xml-final :: *'a xmlst* ⇒ *xml-binds* (-)

-xml-cons :: *xml-bind* ⇒ *xml-binds* ⇒ *xml-binds* (-;//- [*13*, *12*] *12*)
-xml-do :: *ltag* ⇒ *xml-binds* ⇒ *'a* (*XMLdo* (-) {//(*2* -)//} [*1000*,*12*] *1000*)

syntax (*ASCII*)
-xml-take :: *pttrn* ⇒ *'a* ⇒ *xml-bind* ((*2*- <−/ -) *13*)

translations
-xml-block (*-xml-cons* (*-xml-take p x*) (*-xml-final e*))
⇌ *-xml-block* (*-xml-final* (*CONST xml-take x* (λ*p*. *e*)))
-xml-block (*-xml-cons* (*-xml-take-text p*) (*-xml-final e*))
⇌ *-xml-block* (*-xml-final* (*CONST xml-take-text* (λ*p*. *e*)))
-xml-block (*-xml-cons* (*-xml-take-int p*) (*-xml-final e*))
⇌ *-xml-block* (*-xml-final* (*CONST xml-take-int* (λ*p*. *e*)))
-xml-block (*-xml-cons* (*-xml-take-nat p*) (*-xml-final e*))
⇌ *-xml-block* (*-xml-final* (*CONST xml-take-nat* (λ*p*. *e*)))
-xml-block (*-xml-cons* (*-xml-take-att p x*) (*-xml-final e*))
⇌ *-xml-block* (*-xml-final* (*CONST xml-take-attribute x* (λ*p*. *e*)))
-xml-block (*-xml-cons* (*-xml-take-att-optional p x*) (*-xml-final e*))
⇌ *-xml-block* (*-xml-final* (*CONST xml-take-attribute-optional x* (λ*p*. *e*)))
-xml-block (*-xml-cons* (*-xml-take-att-default p d x*) (*-xml-final e*))
⇌ *-xml-block* (*-xml-final* (*CONST xml-take-attribute-default d x* (λ*p*. *e*)))
-xml-block (*-xml-cons* (*-xml-take-optional p x*) (*-xml-final e*))
⇌ *-xml-block* (*-xml-final* (*CONST xml-take-optional x* (λ*p*. *e*)))
-xml-block (*-xml-cons* (*-xml-take-default p d x*) (*-xml-final e*))
⇌ *-xml-block* (*-xml-final* (*CONST xml-take-default d x* (λ*p*. *e*)))
-xml-block (*-xml-cons* (*-xml-take-all p x*) (*-xml-final e*))
⇌ *-xml-block* (*-xml-final* (*CONST xml-take-many 0* ∞ *x* (λ*p*. *e*)))
-xml-block (*-xml-cons* (*-xml-take-many p minOccurs maxOccurs x*) (*-xml-final e*))
⇌ *-xml-block* (*-xml-final* (*CONST xml-take-many minOccurs maxOccurs x* (λ*p*. *e*)))
-xml-block (*-xml-cons* (*-xml-let p t*) *bs*)
⇌ *let p* = *t in -xml-block bs*
-xml-block (*-xml-cons b* (*-xml-cons c cs*))
⇌ *-xml-block* (*-xml-cons b* (*-xml-final* (*-xml-block* (*-xml-cons c cs*))))
-xml-cons (*-xml-let p t*) (*-xml-final s*)
⇌ *-xml-final* (*let p* = *t in s*)
-xml-block (*-xml-final e*) ⇀ *e*
-xml-do t e ⇌ *CONST xml-do t* (*-xml-block e*)

fun *xml-error-to-string* **where**
xml-error-to-string (*Fatal e*) = *String.explode* (*STR ''Fatal*: *''* + *e*)
| *xml-error-to-string* (*TagMismatch e*) = *String.explode* (*STR ''tag mismatch*: *''* + *default-showsl-list showsl-lit e* (*STR ''''*))

definition *parse-xml* :: *'a xmlt* ⇒ *xml* ⇒ *string* +⊥ *'a*
where *parse-xml p xml* ≡
bind2 (*xml-take p xml-return* ([*xml*],[],*False*,[],[]))
(*Left o xml-error-to-string*) *Right*

1.4 Handling of special characters in text

definition *special-map* = *map-of* [
  (''quot'', ''"''), (''#34'', ''"''), — double quotation mark
  (''amp'', ''&''), (''#38'', ''&''), — ampersand
  (''apos'', [*CHR 0x27*]), (''#39'', [*CHR 0x27*]), — single quotes
  (''lt'', ''<''), (''#60'', ''<''), — less-than sign
  (''gt'', ''>''), (''#62'', ''>'') — greater-than sign
]

fun *extract-special*
  where
    extract-special acc [] = *None*
  | *extract-special acc* (*x* # *xs*) =
    (*if x* = *CHR* '';'' *then map-option* (λ*s*. (*s*, *xs*)) (*special-map* (*rev acc*))
    else extract-special (*x*#*acc*) *xs*)

lemma *extract-special-length* [*termination-simp*]:
  assumes *extract-special acc xs* = *Some* (*y*, *ys*)
  shows *length ys* < *length xs*
  using *assms* **by** (*induct acc xs rule*: *extract-special.induct*) (*auto split*: *if-splits*)

fun *normalize-special*
  where
    normalize-special [] = []
  | *normalize-special* (*x* # *xs*) =
      (*if x* = *CHR* ''&'' *then*
        (*case extract-special* [] *xs of*
          None ⇒ ''&'' @ *normalize-special xs*
        | *Some* (*spec*, *ys*) ⇒ *spec* @ *normalize-special ys*)
      else x # *normalize-special xs*)

fun *map-xml-text* :: (*string* ⇒ *string*) ⇒ *xml* ⇒ *xml*
  where
    map-xml-text f (*XML t as cs*) = *XML t as* (*map* (*map-xml-text f*) *cs*)
  | *map-xml-text f* (*XML-text txt*) = *XML-text* (*f txt*)

definition *parse-xml-string* :: 'a *xmlt* ⇒ *string* ⇒ *string* +⊥ 'a
  where
    parse-xml-string p str ≡ *case doc-of-string str of*
    Inl err ⇒ *Left err*
  | *Inr* (*XMLDOC header xml*) ⇒ *parse-xml p* (*map-xml-text normalize-special xml*)

1.5 For Terminating Parsers

primrec *size-xml*
  where *size-xml* (*XML-text str*) = *size str*
  | *size-xml* (*XML tag atts xmls*) = 1 + *size tag* + (∑ *xml* ← *xmls*. *size-xml xml*)

abbreviation *size-xml-state* ≡ *size-xml* ∘ *fst*
abbreviation *size-xmls-state x* ≡ (∑ *xml* ← *fst x*. *size-xml xml*)

lemma *size-xml-nth* [*dest*]: *i* < *length xmls* ⟹ *size-xml* (*xmls*!*i*) ≤ *sum-list* (*map size-xml xmls*)
using *elem-le-sum-list*[*of* - *map Xmlt.size-xml* -, *unfolded length-map*] **by** *auto*

lemma *xml-or-cong*[*fundef-cong*]:
assumes ⋀*info*. *p* (*fst x*, *info*) = *p*′ (*fst x*, *info*)
and ⋀*info*. *q* (*fst x*, *info*) = *q*′ (*fst x*, *info*)
and *x* = *x*′
shows (*p XMLor q*) *x* = (*p*′ *XMLor q*′) *x*′
using *assms*
by (*cases x*, *auto simp*: *xml-or-def intro*!: *Option.bind-cong split*:*sum.split xml-error.split*)

lemma *xml-do-cong*[*fundef-cong*]:
fixes *p* :: ′*a xmlst*
assumes ⋀*tag*′ *atts xmls info*. *fst x* = *XML tag*′ *atts xmls* ⟹ *String.explode tag* = *tag*′ ⟹ *p* (*xmls*,*atts*,*info*) = *p*′ (*xmls*,*atts*,*info*)
and *x* = *x*′
shows *xml-do tag p x* = *xml-do tag p*′ *x*′
using *assms* **by** (*cases x*, *auto simp*: *xml-do-def split*: *xml.split*)

lemma *xml-take-cong*[*fundef-cong*]:
fixes *p* :: ′*a xmlt* **and** *q* :: ′*a* ⇒ ′*b xmlst*
assumes ⋀*a as info*. *fst x* = *a*#*as* ⟹ *p* (*a*, *info*) = *p*′ (*a*, *info*)
and ⋀*a as ret info info*′. *x*′ = (*a*#*as*,*info*) ⟹ *q ret* (*as*, *info*′) = *q*′ *ret* (*as*, *info*′)
and ⋀*info*. *p* (*XML* [] [] [], *info*) = *p*′ (*XML* [] [] [], *info*)
and *x* = *x*′
shows *xml-take p q x* = *xml-take p*′ *q*′ *x*′
using *assms* **by** (*cases x*, *auto simp*: *xml-take-def intro*!: *Option.bind-cong split*: *list.split sum.split*)

lemma *xml-take-many-cong*[*fundef-cong*]:
fixes *p* :: ′*a xmlt* **and** *q* :: ′*a list* ⇒ ′*b xmlst*
assumes *p*: ⋀*n info*. *n* < *length* (*fst x*) ⟹ *p* (*fst x*′ ! *n*, *info*) = *p*′ (*fst x*′ ! *n*, *info*)
and *err*: ⋀*info*. *p* (*XML* [] [] [], *info*) = *p*′ (*XML* [] [] [], *info*)
and *q*: ⋀*ret n info*. *q ret* (*drop n* (*fst x*′), *info*) = *q*′ *ret* (*drop n* (*fst x*′), *info*)
and *xx*′: *x* = *x*′
shows *xml-take-many-sub ret minOccurs maxOccurs p q x* = *xml-take-many-sub ret minOccurs maxOccurs p*′ *q*′ *x*′
proof−
obtain *as b* **where** *x*: *x* = (*as*,*b*) **by** (*cases x*, *auto*)
show *?thesis*
proof (*insert p q*, *fold xx*′, *unfold x*, *induct as arbitrary*: *b minOccurs maxOccurs ret*)
case *Nil*

with *err* **show** *?case* **by** (*cases b*, *auto intro*!: *Option.bind-cong*)
next
case (*Cons a as*)
from *Cons*(*2*,*3*)[**where** *n=0*] *Cons*(*2*,*3*)[**where** *n=Suc n* **for** *n*]
show *?case* **by** (*cases b*, *auto intro*!: *bind2-cong Cons*(*1*) *split*: *sum.split xml-error.split*)
qed
qed

lemma *xml-take-optional-cong*[*fundef-cong*]:
fixes $p :: 'a\ xmlt$ **and** $q :: 'a\ option \Rightarrow 'b\ xmlst$
assumes $\bigwedge a\ as\ info.\ fst\ x = a \# as \Longrightarrow p\ (a, info) = p'\ (a, info)$
and $\bigwedge a\ as\ ret\ info.\ fst\ x = a \# as \Longrightarrow q\ (Some\ ret)\ (as, info) = q'\ (Some\ ret)\ (as, info)$
and $\bigwedge info.\ q\ None\ (fst\ x', info) = q'\ None\ (fst\ x', info)$
and xx': $x = x'$
shows *xml-take-optional p q x* = *xml-take-optional p' q' x'*
using *assms* **by** (*cases x'*, *auto simp*: *xml-take-optional-def split*: *list.split sum.split xml-error.split intro*!: *bind2-cong*)

lemma *xml-take-default-cong*[*fundef-cong*]:
fixes $p :: 'a\ xmlt$ **and** $q :: 'a \Rightarrow 'b\ xmlst$
assumes $\bigwedge a\ as\ info.\ fst\ x = a \# as \Longrightarrow p\ (a, info) = p'\ (a, info)$
and $\bigwedge a\ as\ ret\ info.\ fst\ x = a \# as \Longrightarrow q\ ret\ (as, info) = q'\ ret\ (as, info)$
and $\bigwedge info.\ q\ d\ (fst\ x', info) = q'\ d\ (fst\ x', info)$
and xx': $x = x'$
shows *xml-take-default d p q x* = *xml-take-default d p' q' x'*
using *assms* **by** (*cases x'*, *auto simp*: *xml-take-default-def split*: *list.split sum.split xml-error.split intro*!: *bind2-cong*)

lemma *xml-change-cong*[*fundef-cong*]:
assumes $x = x'$
and $p\ x' = p'\ x'$
and $\bigwedge ret\ y.\ p\ x = Right\ ret \Longrightarrow q\ ret\ y = q'\ ret\ y$
shows *xml-change p q x* = *xml-change p' q' x'*
using *assms* **by** (*cases x'*, *auto simp*: *xml-change-def split*: *option.split sum.split intro*!: *bind2-cong*)

lemma *if-cong-applied*[*fundef-cong*]:
$b = c \Longrightarrow$
$(c \Longrightarrow x\ z = u\ w) \Longrightarrow$
$(\neg\ c \Longrightarrow y\ z = v\ w) \Longrightarrow$
$z = w \Longrightarrow$
$(if\ b\ then\ x\ else\ y)\ z = (if\ c\ then\ u\ else\ v)\ w$
by *auto*

lemma *option-case-cong*[*fundef-cong*]:

option = *option′* ⟹
(*option′* = *None* ⟹ *f1 z* = *g1 w*) ⟹
(⋀*x2*. *option′* = *Some x2* ⟹ *f2 x2 z* = *g2 x2 w*) ⟹
z = *w* ⟹
(*case option of None* ⇒ *f1* | *Some x2* ⇒ *f2 x2*) *z* = (*case option′ of None* ⇒ *g1* | *Some x2* ⇒ *g2 x2*) *w*
by (*cases option*, *auto*)

lemma *sum-case-cong*[*fundef-cong*]:
s = *ss* ⟹
(⋀*x1*. *ss* = *Inl x1* ⟹ *f1 x1 z* = *g1 x1 w*) ⟹
(⋀*x2*. *ss* = *Inr x2* ⟹ *f2 x2 z* = *g2 x2 w*) ⟹
z = *w* ⟹
(*case s of Inl x1* ⇒ *f1 x1* | *Inr x2* ⇒ *f2 x2*) *z* = (*case ss of Inl x1* ⇒ *g1 x1* | *Inr x2* ⇒ *g2 x2*) *w*
by (*cases s*, *auto*)

lemma *prod-case-cong*[*fundef-cong*]: *p* = *pp* ⟹
(⋀*x1 x2*. *pp* = (*x1*, *x2*) ⟹ *f x1 x2 z* = *g x1 x2 w*) ⟹
(*case p of* (*x1*, *x2*) ⇒ *f x1 x2*) *z* = (*case pp of* (*x1*, *x2*) ⇒ *g x1 x2*) *w*
by (*auto split*: *prod.split*)

Mononicity rules of combinators for partial-function command.

lemma *bind2-mono* [*partial-function-mono*]:
assumes *p0*: *mono-sum-bot p0*
assumes *p1*: ⋀*y*. *mono-sum-bot* (*p1 y*)
assumes *p2*: ⋀*y*. *mono-sum-bot* (*p2 y*)
shows *mono-sum-bot* (λ*g*. *bind2* (*p0 g*) (λ*y*. *p1 y g*) (λ*y*. *p2 y g*))
proof (*rule monotoneI*)
fix *f g* :: *′a* ⇒ *′b* +⊥ *′c*
assume *fg*: *fun-ord sum-bot-ord f g*
with *p0* **have** *sum-bot-ord* (*p0 f*) (*p0 g*) **by** (*rule monotoneD*[*of - - - f g*])
then have *sum-bot-ord*
(*bind2* (*p0 f*) (λ *y*. *p1 y f*) (λ *y*. *p2 y f*))
(*bind2* (*p0 g*) (λ *y*. *p1 y f*) (λ *y*. *p2 y f*))
unfolding *flat-ord-def bind2-def* **by** *auto*
also from *p1* **have** ⋀ *y′*. *sum-bot-ord* (*p1 y′ f*) (*p1 y′ g*)
by (*rule monotoneD*) (*rule fg*)
then have *sum-bot-ord*
(*bind2* (*p0 g*) (λ *y*. *p1 y f*) (λ *y*. *p2 y f*))
(*bind2* (*p0 g*) (λ *y*. *p1 y g*) (λ *y*. *p2 y f*))
unfolding *flat-ord-def* **by** (*cases p0 g*) (*auto simp*: *bind2-def*)
also (*sum-bot.leq-trans*)
from *p2* **have** ⋀ *y′*. *sum-bot-ord* (*p2 y′ f*) (*p2 y′ g*)
by (*rule monotoneD*) (*rule fg*)
then have *sum-bot-ord*
(*bind2* (*p0 g*) (λ *y*. *p1 y g*) (λ *y*. *p2 y f*))
(*bind2* (*p0 g*) (λ *y*. *p1 y g*) (λ *y*. *p2 y g*))
unfolding *flat-ord-def* **by** (*cases p0 g*) (*auto simp*: *bind2-def*)

finally (*sum-bot.leq-trans*)
show *sum-bot-ord* (*bind2* (*p0 f*) (λ*y*. *p1 y f*) (λ*y*. *p2 y f*))
(*bind2* (*p0 g*) (λ*ya*. *p1 ya g*) (λ*ya*. *p2 ya g*)) **.**
qed

lemma *xml-or-mono* [*partial-function-mono*]:
assumes *p1*: ⋀*y*. *mono-sum-bot* (*p1 y*)
assumes *p2*: ⋀*y*. *mono-sum-bot* (*p2 y*)
shows *mono-sum-bot* (λ*g*. *xml-or* (λ*y*. *p1 y g*) (λ*y*. *p2 y g*) *x*)
using *p1* **unfolding** *xml-or-def*
by (*cases x*, *auto simp*: *xml-or-def intro*!: *partial-function-mono*,
intro monotoneI, *auto split*: *xml-error.splits simp*: *sum-bot.leq-refl dest*: *monotoneD*[*OF p2*])

lemma *xml-do-mono* [*partial-function-mono*]:
assumes *p1*: ⋀*y*. *mono-sum-bot* (*p1 y*)
shows *mono-sum-bot* (λ*g*. *xml-do t* (λ*y*. *p1 y g*) *x*)
by (*cases x*, *cases fst x*) (*auto simp*: *xml-do-def intro*!: *partial-function-mono p1*)

lemma *xml-take-mono* [*partial-function-mono*]:
assumes *p1*: ⋀*y*. *mono-sum-bot* (*p1 y*)
assumes *p2*: ⋀*x z*. *mono-sum-bot* (λ *y*. *p2 z x y*)
shows *mono-sum-bot* (λ*g*. *xml-take* (λ*y*. *p1 y g*) (λ *x y*. *p2 x y g*) *x*)
proof (*cases x*)
case (*fields a b c d e*)
show *?thesis* **unfolding** *xml-take-def fields split*
by (*cases a*, *auto intro*!: *partial-function-mono p2 p1*)
qed

lemma *xml-take-default-mono* [*partial-function-mono*]:
assumes *p1*: ⋀*y*. *mono-sum-bot* (*p1 y*)
assumes *p2*: ⋀*x z*. *mono-sum-bot* (λ *y*. *p2 z x y*)
shows *mono-sum-bot* (λ*g*. *xml-take-default a* (λ*y*. *p1 y g*) (λ *x y*. *p2 x y g*) *x*)
proof (*cases x*)
case (*fields a b c d e*)
show *?thesis* **unfolding** *xml-take-default-def fields split*
by (*cases a*, *auto intro*!: *partial-function-mono p2 p1*, *intro monotoneI*,
auto split: *xml-error.splits simp*: *sum-bot.leq-refl dest*: *monotoneD*[*OF p2*])
qed

lemma *xml-take-optional-mono* [*partial-function-mono*]:
assumes *p1*: ⋀*y*. *mono-sum-bot* (*p1 y*)
assumes *p2*: ⋀*x z*. *mono-sum-bot* (λ *y*. *p2 z x y*)
shows *mono-sum-bot* (λ*g*. *xml-take-optional* (λ*y*. *p1 y g*) (λ *x y*. *p2 x y g*) *x*)
proof (*cases x*)
case (*fields a b c d e*)
show *?thesis* **unfolding** *xml-take-optional-def fields split*
by (*cases a*, *auto intro*!: *partial-function-mono p2 p1*, *intro monotoneI*,
auto split: *xml-error.splits simp*: *sum-bot.leq-refl dest*: *monotoneD*[*OF p2*])

qed

lemma *xml-change-mono* [*partial-function-mono*]:
assumes *p1*: $\bigwedge y$. *mono-sum-bot* (*p1 y*)
assumes *p2*: $\bigwedge x\ z$. *mono-sum-bot* (λ *y*. *p2 z x y*)
shows *mono-sum-bot* (λg. *xml-change* (λy. *p1 y g*) (λ *x y*. *p2 x y g*) *x*)
unfolding *xml-change-def* **by** (*intro partial-function-mono p1*, *cases x*, *auto intro*: *p2*)

lemma *xml-take-many-sub-mono* [*partial-function-mono*]:
assumes *p1*: $\bigwedge y$. *mono-sum-bot* (*p1 y*)
assumes *p2*: $\bigwedge x\ z$. *mono-sum-bot* (λ *y*. *p2 z x y*)
shows *mono-sum-bot* (λg. *xml-take-many-sub a b c* (λy. *p1 y g*) (λ *x y*. *p2 x y g*) *x*)
proof −
obtain *xs atts allow cands rest* **where** *x*: *x* = (*xs*, *atts*, *allow*, *cands*, *rest*) **by** (*cases x*)
show *?thesis* **unfolding** *x*
proof (*induct xs arbitrary*: *a b c atts allow rest cands*)
case *Nil*
show *?case* **by** (*auto intro*!: *partial-function-mono p1 p2*)
next
case (*Cons x xs*)
show *?case* **unfolding** *xml-take-many-sub.simps*
by (*auto intro*!: *partial-function-mono p2 p1 Cons*, *intro monotoneI*,
auto split: *xml-error.splits simp*: *sum-bot.leq-refl dest*: *monotoneD*[*OF p2*])
qed
qed

partial-function (*sum-bot*) *xml-foldl* :: ('*a* ⇒ '*b xmlt*) ⇒ ('*a* ⇒ '*b* ⇒ '*a*) ⇒ '*a* ⇒ '*a xmlst* **where**
[*code*]: *xml-foldl p f a xs* = (*case xs of* ([],-) ⇒ *Right a*
| - ⇒ *xml-take* (*p a*) (λ *b*. *xml-foldl p f* (*f a b*)) *xs*)

end

theory *Example-Application*
imports
Xmlt
begin

Let us consider inputs that consist of an optional number and a list of first order terms, where these terms use strings as function names and numbers for variables. We assume that we have a XML-document that describes these kinds of inputs and now want to parse them.

definition *exampleInput* **where** *exampleInput* = *STR*
''<*input*>
<*magicNumber*>*42*</*magicNumber*>
<*funapp*> <!−− *first term in list* −−>

```
      <symbol>fo&lt;&gt;bar</symbol>
      <var>1</var> <!-- first subterm -->
      <var>3</var> <!-- second subterm -->
    </funapp>
    <var>15</var> <!-- second term in list -->
  </input>''
```

datatype *fo-term = Fun string fo-term list | Var int*

definition *var :: fo-term xmlt* **where** *var = xml-change (xml-int (STR ''var'')) (xml-return ∘ Var)*

a recursive parser is best defined via partial-function. Note that the xml-argument should be provided, otherwise strict evalution languages will not terminate.

```
partial-function (sum-bot) parse-term :: fo-term xmlt
where
 [code]: parse-term xml = (
    XMLdo (STR ''funapp'') {
      name ← xml-text (STR ''symbol'');
      args ←∗ parse-term;
      xml-return (Fun name args)
    } XMLor var) xml
```

for non-recursive parsers, we can eta-contract

```
definition parse-input :: (int option × fo-term list) xmlt where
  parse-input = XMLdo (STR ''input'') {
      onum ←? xml-int (STR ''magicNumber'');
      terms ←∗ parse-term;
      xml-return (onum,terms)
   }
```

definition *test* **where** *test = parse-xml-string parse-input (String.explode exampleInput)*

```
value test
export-code test checking SML
end
```