Language Module

smlhelp.github.io – Auxiliary Library

Jacob Neumann

July 2021

Contents

1 Decision Problems

Defn. 1 (Set of values)

For any type `t`, write $\mathsf{Values}(\texttt{t})$ to denote the set of all values of type `t` (up to extensional equivalence).

Example 2

- $\mathsf{Values}(\texttt{bool}) = \{\texttt{true}, \texttt{false}\}$
- $\mathsf{Values}(\texttt{bool -> bool}) = \{\texttt{not}, \texttt{Fn.id}, \texttt{(fn \_ => true)}, \texttt{(fn \_ => false)}\}$

Defn. 3

A **decision problem** of type `t` is a subset

$$L \subseteq \mathsf{Values}(\texttt{t})$$

Defn. 4

A decision problem $L \subseteq \mathsf{Values}(\texttt{t})$ is said to be **decidable** if there is a **total** value `D : t -> bool` such that

$$\texttt{D(v)} \Longrightarrow \texttt{true} \quad \text{iff} \quad \texttt{v} \in L$$

for all values `v:t`. In this case, we say that `D` **decides** (or **computes**) L

Example 5

The *subset sum problem* is a family of decision problems of type `int list`, one for each value `n : int`

$$\mathsf{SUBSETSUM}_{\texttt{n}} = \{\texttt{l} \in \mathsf{Values}(\texttt{int list}) \mid \texttt{foldr op+ 0 l} \cong \texttt{n}\}$$

For each `n`, the problem $\mathsf{SUBSETSUM}_{\texttt{n}}$ is decidable: we can define a total function

```
subsetSum n : int list -> bool
```

such that `subsetSum n l` $\Longrightarrow$ `true` iff `foldr op+ 0 l` $\cong$ `n`.

Defn. 6

Given a total function `D : t -> bool`, define the **language** of `D` to be the set

$$\mathcal{L}(\texttt{D}) = \{\texttt{v} \in \mathsf{Values}(\texttt{t}) \mid \texttt{D(v)} \Longrightarrow \texttt{true}\}.$$

Defn. 7

An **equality type** is any type `t` such that

```
(op =) : t * t -> bool
```

is well-typed, i.e. any type whose values we can compare with the `=` and `<>` operators.

Defn. 8

Given an equality type `Sigma`, a **decision problem over the alphabet `Sigma`** is a subset

$$L \subseteq \mathsf{Values}(\texttt{Sigma list}).$$

2 Specification

Note 9

Throughout, `Sigma` will denote some equality type, the type of our "alphabet". Though some of our results will hold when `Sigma` is allowed to be a general polymorphic type (e.g. Lemma 14), we are mainly concerned with situations where `Sigma` is an equality type (and the values `singleton` and `just` demand that `Sigma` be an equality type).

A paradigm example is to take

```
type Sigma = char
```

Lemma 10

For any total function `D : Sigma list -> bool` and any subset $L \subseteq \mathsf{Values}(\texttt{Sigma list})$, the following are equivalent:

(1) `D` computes L (Defn. 4)

(2) For *all* values `v : Sigma list`,

$$\texttt{D(v)} \quad \Longrightarrow \quad \begin{cases} \texttt{true} & \text{if } \texttt{v} \in L \\ \texttt{false} & \text{if } \texttt{v} \notin L \end{cases}$$

1. $\mathcal{L}(\texttt{D}) = L$ (Defn. 6)

Type Spec

```
'S language = 'S list -> bool
```

Value Spec

`everything : 'S language`

ENSURES: `everything : Sigma language` is a total function computing the decision problem Values(`Sigma list`) over the alphabet `Sigma`.

Value Spec

`nothing : 'S language`

ENSURES: `nothing : Sigma language` is a total function computing the decision problem $\emptyset$.

Value Spec

`singleton : ''S list -> ''S language`

ENSURES: `(singleton v): Sigma language` is a total function computing the decision problem $\{\mathtt{v}\}$

Value Spec

`just : ''S list list -> ''S language`

ENSURES: `(just` $[\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_n]$`): Sigma language` is a total function computing the decision problem $\{\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_n\}$

Value Spec

`Or : 'S language * 'S language -> 'S language`

REQUIRES: `L1` and `L2` are total

ENSURES: `Or(L1,L2)` is a total function such that

$$\mathcal{L}(\mathtt{Or(L1,L2)}) = \mathcal{L}(\mathtt{L1}) \cup \mathcal{L}(\mathtt{L2})$$

Value Spec

`And : 'S language * 'S language -> 'S language`

REQUIRES: `L1` and `L2` are total

ENSURES: `And(L1,L2)` is a total function such that

$$\mathcal{L}(\mathtt{And(L1,L2)}) = \mathcal{L}(\mathtt{L1}) \cap \mathcal{L}(\mathtt{L2})$$

Value Spec

`Not : 'S language -> 'S language`

REQUIRES: `L` is total

ENSURES: For any `L : Sigma language`, `Not(L)` is a total function such that

$$\mathcal{L}(\texttt{Not(L)}) = \mathsf{Values}(\texttt{Sigma list}) \setminus \mathcal{L}(\texttt{L})$$

Value Spec

`Xor : 'S language * 'S language -> 'S language`

REQUIRES: `L1` and `L2` are total

ENSURES: `Xor(L1,L2)` is a total function such that

$$\mathcal{L}(\texttt{Xor(L1,L2)}) = \mathcal{L}(\texttt{Or(L1,L2)}) \setminus \mathcal{L}(\texttt{And(L1,L2)})$$

Value Spec

`lengthEqual : int -> 'S language`

REQUIRES: $\texttt{n} \geq 0$

ENSURES: `lengthEqual n` is a total function such that

$$\mathcal{L}(\texttt{lengthEqual n}) \\ = \{\texttt{s} \in \mathsf{Values}(\texttt{Sigma list}) \mid \texttt{((List.length s)=n)} \implies \texttt{true}\}$$

Value Spec

`lengthLess : int -> 'S language`

REQUIRES: $\texttt{n} \geq 0$

ENSURES: `lengthLess n` is a total function such that

$$\mathcal{L}(\texttt{lengthLess n}) \\ = \{\texttt{s} \in \mathsf{Values}(\texttt{Sigma list}) \mid \texttt{((List.length s)<n)} \implies \texttt{true}\}$$

Value Spec

`lengthGreater : int -> 'S language`

REQUIRES: $\texttt{n} \geq 0$

ENSURES: `lengthGreater n` is a total function such that

$$\mathcal{L}(\texttt{lengthGreater n}) \\ = \{\texttt{s} \in \mathsf{Values}(\texttt{Sigma list}) \mid \texttt{((List.length s)>n)} \implies \texttt{true}\}$$

3 Implementation

aux-library: Language.sml

```
type 'S language = 'S list -> bool

fun everything (x:'S list) = true
fun nothing (x : 'S list) = false

val singleton = Fn.equal
fun just ([] : ''S list list) s = false
  | just (x::xs) s = (s=x) orelse just xs s

fun Or (L1,L2) s = (L1 s) orelse (L2 s)
fun And (L1,L2) s = (L1 s) andalso (L2 s)
fun Not L = not o L
fun Xor (L1,L2) s = (L1 s) <> (L2 s)

fun lengthEqual n s = (List.length s)=n
fun lengthLess n s = (List.length s)<n
fun lengthGreater n s = (List.length s)>n

fun str L = L o String.explode
```

4 Lemmas

Throughout, `L,L1,L2,L3 : Sigma language` are total.

Prop 11 (Totality)

All values of the `Sigma language` type produced using the `Language` module methods are total:

- `everything` is total
- `nothing` is total
- For any value `v : Sigma list`, `(singleton v)` is total
- For any value `l : Sigma list list`, `(just l)` is total
- All the curried higher-order functions (`singleton`, `just`, `Or`, `And`, `Not`, `Xor`, `lengthEqual`, `lengthLess`, `lengthGreater`, and `str`) are all *total* in the trivial sense: upon being supplied one argument, they evaluate to a value (a function expecting the next curried argument)
- If `L1,L2 : Sigma language` are total,
  - `Or(L1,L2)` is total
  - `And(L1,L2)` is total
  - `Not(L1)` is total
  - `Xor(L1,L2)` is total
- For any `n` $\geq$ `0`,
  - `lengthEqual n` is total
  - `lengthLess n` is total
  - `lengthGreater n` is total
- If `L : char language` is total, so too is `str L`

Proof. We prove several paradigmatic cases, and the others can be done similarly.

Recall `just` is implemented as

aux-library: Language.sml

```
fun just ([] : ''S list list) s = false
  | just (x::xs) s = (s=x) orelse just xs s
```

So we can prove the totality of `(just l)` by structural induction on `l : Sigma list list`. The base case is immediate: for any value `s : Sigma list`, `just [] s` $\Longrightarrow$

`false`, a value. Inductively assuming `just xs s` valuable, then we can see that `(just (x::xs) s)` is also valuable, since `(s=x)` is valuable and, if `(s=x)` evaluates to `false`, then

```
just (x::xs) s   ⟹   just xs s
```

which our IH tells us is valuable, completing the proof.

Recall `And` is implemented as

aux-library: Language.sml

```
fun And (L1,L2) s = (L1 s) andalso (L2 s)
```

so if `L1` and `L2` are total, then `(L1 s)` and `(L2 s)` are valuable, hence `(L1 s) andalso (L2 s)` is valuable, proving `And(L1,L2)` total.

Taking for granted that `List.length` is total, it follows that the expression `(List.length s)=n` is valuable. Since this is the body of `lengthEqual n s`, we get that `lengthEqual n` is total.

Proving the totality of `str L` from the totality of `L` is a straightforward consequence of the totality of `String.explode`. □

Lemma 12

```
just  ≅  (foldr Or nothing) o (map singleton)
```

Proof. It suffices to show that for all values `l : Sigma list list`,

```
just l  ≅  foldr Or nothing (map singleton l)
```

which we do by structural induction on `l`.

BC `l = []`. Pick arbitrary `s : Sigma list`. Then

```
just [] s
  ≅  false                                     (Defn. just)
  ≅  nothing s                                 (Defn. nothing)
  ≅  (foldr Or nothing []) s                   (Defn. foldr)
  ≅  (foldr Or nothing (map singleton [])) s   (Defn. map)
```

Establishing that `just []` ≅ `foldr Or nothing (map singleton [])`.

IH Suppose for some `xs : Sigma list list` that

```
just xs  ≅  foldr Or nothing (map singleton xs)
```

Now pick some `x : Sigma list`. We'll show that

```
just (x::xs)  ≅  foldr Or nothing (map singleton (x::xs))
```

Pick arbitrary `s : Sigma list`.

```
just (x::xs) s
≅ (s=x) orelse just xs s                                        (Defn. just)
≅ (x=s) orelse just xs s                                        (Symmetry of =)
≅ (Fn.equal x s) orelse just xs s                               (Defn. Fn.equal)
≅ (singleton x s) orelse just xs s                              (Defn. singleton)
≅ (singleton x s) orelse (foldr Or nothing (map singleton xs)) s
                                                                IH
≅ Or(singleton x, (foldr Or nothing (map singleton xs))) s
      (Defn. Or, Prop. 11, higher-order totality of map and foldr)
≅ (foldr Or nothing ((singleton x)::(map singleton xs))) s
      (Defn. foldr, Prop. 11, higher-order totality of map)
≅ (foldr Or nothing (map singleton (x::xs))) s                  (Defn. map)
```

so we're done. □

Cor. 13

`nothing ≅ just []`

Lemma 14

For any total `L,L1,L2,L3 : Sigma language`, the following equivalences hold

- `And(L1,And(L2,L3))  ≅  And(And(L1,L2),L3)`
- `And(L1,L2)  ≅  And(L2,L1)`
- `Or(L1,Or(L2,L3))  ≅  Or(Or(L1,L2),L3)`
- `Or(L1,L2)  ≅  Or(L2,L1)`
- `And(L,everything)  ≅  L  ≅  And(everything,L)`
- `Or(L,nothing)  ≅  L  ≅  Or(nothing,L)`
- `Not(Not(L))  ≅  L`
- `Or(L1,And(L1,L2))  ≅  L1  ≅  And(L1,Or(L1,L2))`
- `Or(L1,And(L2,L3))  ≅  And(Or(L1,L2),Or(L1,L3))`
  `And(L1,Or(L2,L3))  ≅  Or(And(L1,L2),And(L1,L3))`

- `And(L,Not L)` $\cong$ `nothing`
  `Or(L,Not L)` $\cong$ `everything`

Lemma 15

For all values `n` $\geq$ `0`,

`Not(lengthGreater n)` $\cong$ `Or(lengthEqual n,lengthLess n)`

Proof. Pick arbitrary `s : Sigma list`, and let `m` denote the value of `List.length s`.

```
(Not(lengthGreater n)) s
≅ not(lengthGreater n s)                                  (Defn. Not, Prop. 11)
≅ not((List.length s)>n                                   (Defn. lengthGreater)
≅ not (m>n)                                               (Defn. m)
≅ m=n orelse m<n                                          (math)
≅ ((List.length s)=n) orelse ((List.length s)<n)          (Defn. m)
≅ (lengthEqual n s) orelse (lengthLess n s)
                                   (Defn. lengthEqual and lengthLess)
≅ (Or(lengthEqual n, lengthLess n)) s                     (Defn. Or, Prop. 11)
```

as desired. □

This proof can be modified to prove similar equivalences, e.g.

`Not(lengthLess n)` $\cong$ `Or(lengthEqual n,lengthGreater n).`