Set types

The set types are one of the compound types available in SOFL, and usually used for the abstraction of data items that have a collection of elements.

The outline of this part:

- What is a set?
- Set type constructors
- Constructors and operators on sets
- Specification with set types

What is a set?

A set is an unordered collection of distinct objects where each object is known as an element of the set.

For example:

(1) A class is a set of students.

(2) A car park is a set of cars.

A set of values is enclosed with braces. For example,

(1) {5, 9, 10}

(2) {"John", "Chris", "David", "Jeff"}

(3) {"Java", "Pascal", "C", "C++", "Fortran"}

Notice: {a, a, b} is not a legal set.

Set type declaration

Let T be an arbitrary type, A be a set type to be defined. Then, the declaration of A has the form:

A = set of T

where T is called "element type".

Formally, A is the power set of T:

$A = \{x \mid subset(x, T)\}$

where subset(x, T) means that x is a subset of T.

For example: let A be defined as follows:

type

A = set of {<DOG>, <CAT>, <COW>}

This means:

A = {{ }, {<DOG>}, {<CAT>}, {<COW>},
{<DOG>, <CAT>}, {<DOG>, <COW>},
{<CAT>, <COW>}, {<DOG>, <CAT>, <COW>}}

Set variable declaration:

Let s be a variable of type A, which is declared as:

s: A;

then, s can take any value of A:

s = { } (empty set) or
s = {<DOG>} or
s = {<CAT>} or
s = {<COW>} or
s = {<DOG>, <CAT>} or
s = {<DOG>, <COW>} or
s = {<CAT>, <COW>} or
s = {<DOG>, <CAT>, <COW>}

Constructors and operators on sets

1. Constructors

A constructor of set types is a special operator that constitutes a set value from the elements of an element type.

There are two set constructors:

set enumeration and set comprehension.

A set enumeration has the format:

$\{e\_1, e\_2, ..., e\_n\}$

where e_i (i=1..n) are the elements of the set {e_1, e_2, ..., e_n}.

Examples:

{5, 9, 10, 50}

{'a', 't', 'l'}

A set comprehension has the form:

$\{e(x_1, x_2, ..., x_n) \mid x_1: T_1, x_2: T_2, ..., x_n: T_n \ \& \ P(x_1, x_2, ..., x_n)\}$

or

$\{e(x_1, x_2, ..., x_n) \mid P(x_1, x_2, ..., x_n)\}$

where $n >= 1$.

The set comprehension defines a collection of values resulting from evaluating the expression $e(x_1, x_2, ..., x_n)$ $(n>=1)$ under the condition that the involved variables $x_1, x_2, ..., x_n$ take values from sets (or types) $T_1, T_2, ..., T_n$, respectively, and satisfies property $P(x_1, x_2, ..., x_n)$.

Examples:

$\{x \mid x: nat \ \& \ 1 < x < 5\} = \{2, 3, 4\}$

$\{y \mid y: nat0 \ \& \ y <= 5\} = \{0, 1, 2, 3, 4, 5\}$

$\{x + y \mid x: nat0, y: nat0 \ \& \ 1 < x + y < 8\} = \{2, 3, 4, 5, 6, 7\}$

$\{i \mid i \ inset \ \{1, 3, 5, 7\}\} = \{1, 3, 5, 7\}$

We can also use the following special notation to represent a set containing an interval of integers:

$\{i, ..., k\} = \{j \mid j: int \ \& \ i <= j <= k\}$

Thus:

$\{1, ..., 5\} = \{1, 2, 3, 4, 5\}$

$\{-2, ..., 2\} = \{-2, -1, 0, 1, 2\}$

2. Operators

2.1 Membership (inset)

inset: T * set of T --> bool

Examples:

7 inset {4, 5, 7, 9} <=> true

3 inset {4, 5, 7, 9} <=> false

2.2 Non-membership (notin)

notin: T * set of T --> bool

Examples:

7 notin {4, 5, 7, 9} <=> false

3 notin {4, 5, 7, 9} <=> true

2.3 Cardinality (card)

card: set of T --> nat0

Examples:

card({5, 7, 9}) = 3

card({'h', 'o', 's', 'e', 'i'}) = 5

2.4 Equality and inequality (=, <>)

=: set of T * set of T --> bool

s1 = s2

== forall[x: s1] | x inset s2 and card(s1) = card(s2)

"==" means "defined as".

<>: set of T * set of T --> bool

s1 <> s2

== (exists[x: s1] | x notin s2) or (exists[x: s2] | x notin s1)

Examples:

{5, 15, 25} = { 25, 15, 5} <=> true

{5, 15, 25} <> {5, 20, 30} <=> true

2.5 Subset (subset)

subset: set of T * set of T --> bool

subset(s1, s2) == forall[x: s1] | x inset s2

Examples:

Let s1 = {5, 15, 25}, s2 = {5, 10, 15, 20, 25, 30}.
Then:

subset(s1, s2) <=> true   subset(s2, s1) <=> false
subset({ }, s1) <=> true   subset(s1, s1) <=> true

2.6 Proper subset (psubset)

psubset: set of T * set of T --> bool

psubset(s1, s2) == subset(s1, s2) and s1 <> s2

Examples:

let s1 = {5, 15, 25} and s2 = {5, 10, 15, 25, 30}.

Then:

psubset(s1, s2) <=> true

psubset(s1, s1) <=> false

psubset(s2, s1) <=> false

psubset({ }, s1) <=> true

2.7 Member access (get)

get: set of T --> T

get(s) == if s <> { } then x else nil

where x inset s.

Examples: assume s = {5, 15, 25}, then

get(s) = 5 or

get(s) = 15 or

get(s) = 25

And s still remains the same as before:

s = {5, 15, 25}.

2.8 Union (union)

union: set of T * set of T --> set of T

union(s1, s2) == {x | x inset s1 or x inset s2}

Examples:

union({5, 15, 25}, {15, 20, 25, 30}) =
{5, 15, 25, 20, 30}

union({15, 20, 25, 30}, {5, 15, 25}) =
{5, 15, 25, 20, 30}

The union operator is commutative. Thus,

union(s1, s2) = union(s2, s1).

It is also associative, that is,

union(s1, union(s2, s3)) =
    union(union(s1, s2), s3).

Due to these properties, the operator union can be extended to deal with more than two sets:

union: set of T * set of T * ... * set of T --> set of T

union(s1, s2, ..., sn) == {x | x inset s1 or x inset s2 or ... or x inset sn}

2.9 Intersection (inter)

inter: set of T * set of T --> set of T

inter(s1, s2) == {x | x inset s1 and x inset s2}

For example, let s1 = {5, 7, 9},

s2 = {7, 10, 9, 15},

s3 = {8, 5, 20}.

Then

inter(s1, s2) = {7, 9}

inter(s1, s3) = {5}

inter(s2, s3) = { }

The inter operator is commutative and associative. That is,

inter(s1, s2) = inter(s2, s1),

inter(s1, inter(s2, s3)) = inter(inter(s1, s2), s3).

We can also extend the inter operator to deal with more than two operands:

inter: set of T * set of T * ... * set of T --> set of T

inter(s1, s2, ..., sn)

== {x | x inset s1 and x inset s2 and ... and x inset sn}

2.10 Difference (diff)

```
diff: set of T * set of T --> set of T
diff(s1, s2) == {x | x inset s1 and x notin s2}
```

For example, let s1 = {5, 7, 9}
s2 = {7, 10, 9, 15}
s3 = {8, 12}.

Then

```
diff(s1, s2) = {5}
diff(s1, s3) = {5, 7, 9}
diff(s2, s1) = {10, 15}
diff(s1, { }) = s1
```

2.11 Distributed union (dunion)

A set can be a set of sets, and the distributed union of such a set is an operation that obtains the union of all the member sets of the set.

```
dunion: set of set of T --> set of T
dunion(s) == union(s1, s2, ..., sn)
```

where s = {s1, s2, ..., sn}.

Example:

Let s1 = {{5, 10, 15}, {5, 10, 15, 25}, {10, 25, 35}}
Then
  dunion(s1) = union({5, 10, 15}, {5, 10, 15, 25}, {10, 25, 35}}
  = {5, 10, 15, 25, 35}

2.12 Distributed intersection (dinter)

dinter: set of set of T --> set of T

dinter(s) == inter(s1, s2, ..., sn)

where s = {s1, s2, ..., sn}.

For example, let

s = {{5, 10, 15}, {5, 10, 15, 25}, {10, 25, 35}}.

Then

dinter(s) = inter({5, 10, 15}, {5, 10, 15, 25},
{10, 25, 35}} = {10}

2.13 Power set (power)

Given a set, we can apply the operator power to yield its power set that contains all the subsets of the set, including the empty set.

```
power: set of T --> set of set of T
power(s) == { s1 | subset(s1, s)}
```

Example: let s = {5, 15, 25}. Then

power(s) = {{ }, {5}, {15}, {25}, {5, 15}, {15, 25}, {5, 25}, {5, 15, 25}}

Specification with set types

An Email_Address_Book

```
module Email_Address_Book;
 type
  EmailAddress = given;
 var
  email_book: set of EmailAddress;
 behav: CDFD_1;
```

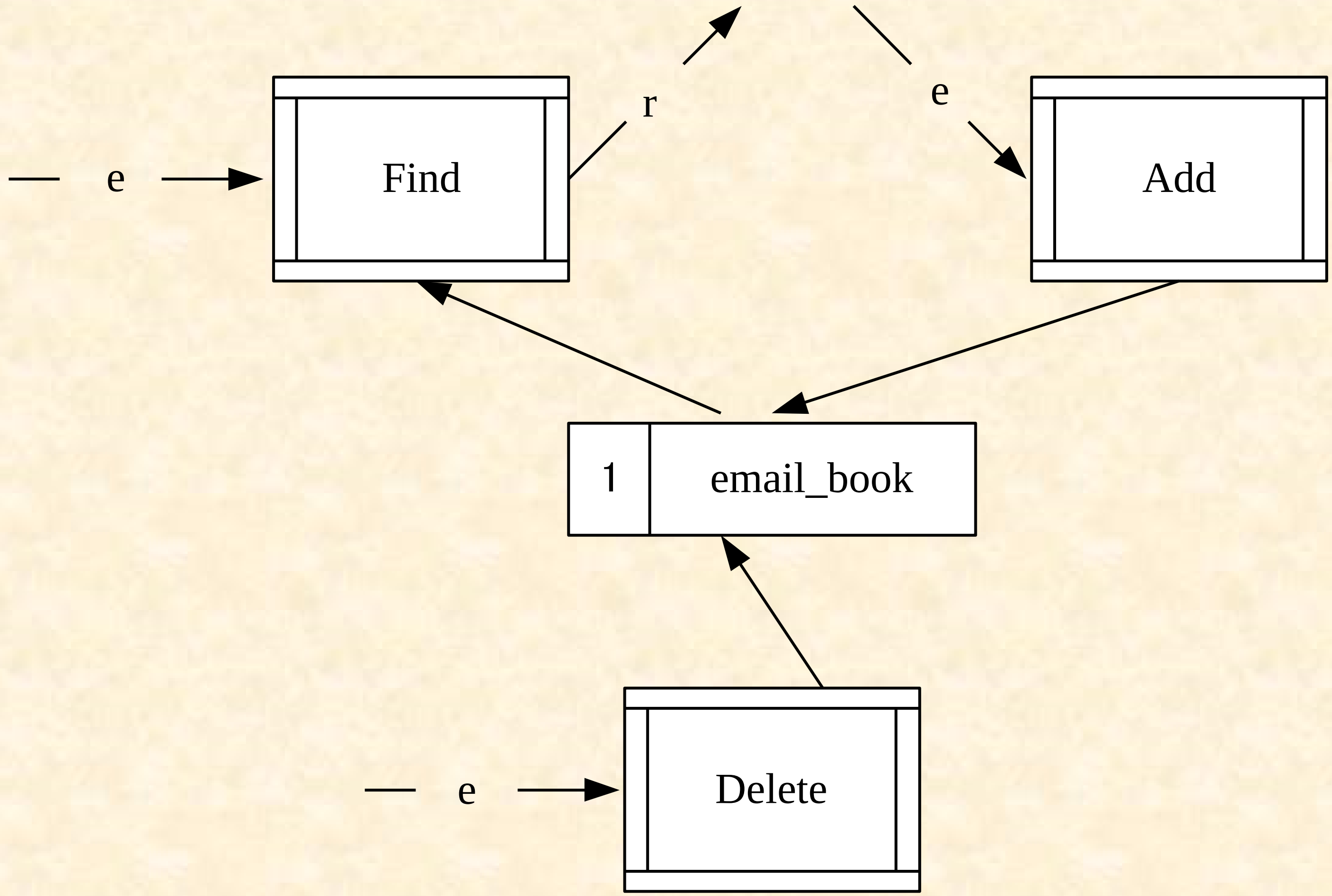

Figure 1

```
process Find(e: EmailAddress) r: bool
ext rd email_book
post r = (e inset email_book)
end_process;

process Add(e: EmailAddress)
ext wr email_book
pre e notin email_book
post email_book = union(~email_book, {e})
end_process;

process Delete(e: EmailAddress)
ext wr email_book
post email_book = diff(~email_book, {e})
end_process;
end_module;
```

Class exercise 5

1. Let s1 = {5, 15, 25}, s2 = {15, 30, 50},
   s3 = {30, 2, 8}, and s = {s1, s2, s3}.

   Evaluate the expressions:

a. card(s1)

b. card(s)

c. union(s1, s2)

d. diff(s2, s3)

e. inter(union(s2, s3), s1)

f. dunion(s)

g. dinter(s)

h. inter(union(s1, s3), diff(s2, union(s1, s3))

2. Construct a module to model a telephone book containing a set of telephone numbers. The necessary processes are Add, Find, Delete, and Update. The process Add adds a new telephone number to the book; Find tells whether a given telephone number is available or not in the book; Delete eliminates a given telephone number from the book; and Update replaces an old telephone number with a new number in the book.

3. Write a specification for a process Merge. The process takes two groups of students and merge them into one group. Since the merged group will be taught by a different professor, the students from both groups may drop from the merged group (but exactly which students will drop is unknown).