CTEC2901 Data Structures and Algorithms 2013-14

Sets

David Smallwood

Contents

1 Description

A set is a collection of items in which there are no duplicates and in which the order is not significant. Note that the lack of ordering is at the abstract level: there is no notion of the "first" item in a set; there is no specific point of interest within a set; there is no significance to the order in which items are added and removed. A set data structure exhibits the external behaviour of a mathematical set. However, this does not mean that sets cannot take advantage of any known ordering between elements. A set data structure may be implemented as a balanced binary tree, or as a hash table, for example, in order to gain efficient search times.

1.1 A Set in Haskell

The operations on a set can be specified and implemented using a very high level language like Haskell. For demonstration purposes we will choose to implement the set using a linear list data structure but this does not give efficient search time characteristics. Nevertheless our example will serve to demonstrate the features of a set and to show how one data structure can be realised in terms of another.

The module definition and export list provides the public interface:

```
> module Set (
>   Set         -- The abstract data type
> , emptyset    -- A set with no elements
```

```
> , isEmpty     -- True if the given set is empty
> , isSubset    -- True if first set is a strict subset of second set
> , isEqualTo   -- True if both sets contain exactly the same elements
> , isSubsetEq  -- True if first set is either a strict subset of or is equal to second set
> , count       -- Number of elements in the set
> , isIn        -- Set membership
> , insertInto  -- Insert a single item into the set
> , removeFrom  -- Remove a single item from the set
> , intersect   -- Return a new set containing common elements from both sets
> , union       -- Return a new set containing all elements from both sets
> , minus       -- Return a new set containing all elements from first set not in second set
> , powerset    -- Return a new set containing all the subsets of the given set
> ) where

> import Data.List (intersperse)

> infixr 7 `insertInto`, `removeFrom`
> infixl 6 `intersect`, `union`, `minus`
> infixl 5 `isIn`
```

We will define the new data type *Set* α in terms of a list $[\alpha]$, and collect the type signatures of all the methods together:

```
> data Set a = Set [a]

> emptyset     :: Eq a => Set a
> isEmpty      :: Eq a => Set a -> Bool
> isSubset     :: Eq a => Set a -> Set a -> Bool
> isEqualTo    :: Eq a => Set a -> Set a -> Bool
> isSubsetEq   :: Eq a => Set a -> Set a -> Bool
> count        :: Eq a => Set a -> Int
> isIn         :: Eq a => a -> Set a -> Bool
> insertInto   :: Eq a => a -> Set a -> Set a
> removeFrom   :: Eq a => a -> Set a -> Set a
> intersect    :: Eq a => Set a -> Set a -> Set a
> union        :: Eq a => Set a -> Set a -> Set a
> minus        :: Eq a => Set a -> Set a -> Set a
> powerset     :: Eq a => Set a -> Set (Set a)
```

Now we will implement each of the set methods in turn:

Empty Set Mathamatically $\{\}$ (or $\varnothing$). This is implemented directly using an empty list.

```
> emptyset = Set []
```

isEmpty Mathamatically $(\lambda\, s : \mathit{Set}\ \alpha \to s = \varnothing)$. The set is empty if there are no elements in the underlying list.

```
> isEmpty (Set xs) = null xs
```

isSubset s t Mathamatically $s \subset t$. This is easily implemented by delegation.

```
> s `isSubset` t = s `isSubsetEq` t && not (s `isEqualTo` t)
```

isEqualTo s t Mathamatically $s = t$. This is true only when all the elements in the underlying lists are each contained within the other.

```
> Set xs `isEqualTo` Set ys = all (`elem` ys) xs && all (`elem` xs) ys
```

isSubsetEq s t Mathamatically $s \subseteq t$. This is true only when all the elements in the the first list are each contained within the second list.

```
> Set xs `isSubsetEq` Set ys = all (`elem` ys) xs
```

count s Mathamatically $\#s$. The size of the set is equal to the length of the underlying list because the list does not store duplicate values.

```
> count (Set xs) = length xs
```

isIn x s Mathamatically $x \in s$. This is true if the value is contained within the underlying list.

```
> x `isIn` Set xs = x `elem` xs
```

insertInto x s Mathamatically $s \cup \{x\}$. The value x is added to the set. Note that if the set already contains x then the resulting set is equal to s.

```
> x `insertInto` Set xs | x `elem` xs = Set xs
>                       | otherwise   = Set (x:xs)
```

removeFrom x s Mathamatically $s \setminus \{x\}$. The value x is added to the set. Note that if the set already contains x then the resulting set is equal to s.

```
> x `removeFrom` Set xs  = Set (remove x xs)
>    where remove x []     = []
>          remove x (y:ys) | x==y      = ys
>                          | otherwise = y:remove x ys
```

intersect s t Mathamatically $s \cap t$. The intersection of the sets contains all the elements from s that are also in t.

```
> Set xs `intersect`  Set ys = Set (filter (`elem` ys) xs)
```

union s t Mathamatically $s \cup t$. The union of the sets contains all the elements from s and also all the *extra* (i.e. not in s) elements from t.

```
> Set xs `union`  Set ys = Set (xs ++ filter (`notElem` xs) ys)
```

minus s t Mathamatically $s \setminus t$. The difference of the sets contains all the elements from s that are not elements in t.

```
> Set xs `minus` Set ys = Set (filter (`notElem` ys) xs)
```

powerset s Mathamatically $\mathbb{P}\, s$. This requires the generation of all the subsets of s.

```
> powerset s = subs s
>  where
>    subs (Set [])     = emptyset `insertInto` emptyset
>    subs (Set (x:xs)) = setmap (x `insertInto`) subsxs `append` subsxs
>      where
>        subsxs = subs (Set xs)
>        setmap f (Set xs) = Set (map f xs)
>        Set xs `append` Set ys = Set (xs ++ ys)
```

Finally we overload the $==$ and $<$ operators so that they can be used with our new set data type. A simple *pretty-print* overloading of *show* is also provided.

```
> instance Eq a   => Ord  (Set a) where s <= t = s `isSubsetEq` t

> instance Eq a   => Eq   (Set a) where s == t = s `isEqualTo` t

> instance Show a => Show (Set a) where
>   show (Set xs) = "{" ++ concat (intersperse ", " (map show xs)) ++ "}"
```

1.2 A Haskell Session with the Set abstraction

We can demonstrate the use of the set data strucutre using a Haskell session.

```
*Set> :load Set.lhs
[1 of 1] Compiling Set              ( Set.lhs, interpreted )
Ok, modules loaded: Set.

*Set> let s = 1 `insertInto` 2 `insertInto` 3 `insertInto` emptyset
s :: Set Integer
*Set> s
{1, 2, 3}
it :: Set Integer

*Set> let t = 2 `insertInto` 2 `removeFrom` s
t :: Set Integer
*Set> t
{2, 1, 3}
it :: Set Integer

*Set> let u = 3 `insertInto` 4 `insertInto` 5 `insertInto` emptyset
u :: Set Integer
*Set> u
{3, 4, 5}
it :: Set Integer
*Set> s `union` u
{1, 2, 3, 4, 5}
it :: Set Integer
*Set> s `union` u `minus` t
{4, 5}
it :: Set Integer

*Set> powerset s
{{1, 2, 3}, {1, 2}, {1, 3}, {1}, {2, 3}, {2}, {3}, {}}
it :: Set (Set Integer)
*Set> powerset t
{{2, 1, 3}, {2, 1}, {2, 3}, {2}, {1, 3}, {1}, {3}, {}}
it :: Set (Set Integer)
*Set> powerset s == powerset t
True
it :: Bool
```

2 A Set Data Structure in C

A notable point about the set data structure in Haskell (previous section) is that the sets are *immutable*. It is uncommon to implement immutable data structures in C – the lack of a garbage collector makes it difficult – so we will develop a *mutable* set data structure.

The underlying implementation will make use of our existing circular list library.

Below we include a header file. Note that the intersect, union, and minus operations (along with insert and remove) are *destructive*: the first parameter will be updated to reflect the change that occurs through the operation - such data strucutres are *mutable*.

```
// A set data structure
// Author: drs

#ifndef SET_H
#define SET_H

#include "any.h"

typedef struct set_implementation set;
typedef int  (*equals)(any x, any y);  // returns one if the items are equal, else zero
typedef void (*printer)(any x);        // outputs item x on stdout

set* new_set           (printer item_printer, equals item_compare);
int  set_isempty       (set *s);
int  set_isSubset      (set *s, set * t);
int  set_isEqualTo     (set *s, set * t);
int  set_isSubsetEq    (set *s, set * t);
int  set_count         (set *s);
int  set_isin          (set *s, any x);
void set_insertInto    (set *s, any x);     // s = s u {x}
void set_removeFrom    (set *s, any x);     // s = s \ {x}
void set_intersectWith (set *s, set * t);   // s = s n t    t is unchanged
void set_unionWith     (set *s, set * t);   // s = s u t    t is unchanged
void set_minusWith     (set *s, set * t);   // s = s \ t    t is unchanged
set* set_powerset      (set *s);            // generates new set
void set_print         (set *s);
void set_release       (set *s);

int  seteq             (any s, any t);      // in case of creating sets of sets
void setprn            (any s);             // these cover functions please the typechecker

#endif
```

Practical Exercise 1

Complete the set data structure using the header file provided in these notes. A partial solution is given below.

NB:

1. The *seteq* and *setprn* functions are simply covers for *set_isEqualTo* and *set_print* in which the parameters are given using the *any* data type. This is because when a *set of sets* is required then these methods match the signature required by the constructor.

2. The *powerset* function is difficult – you may wish to leave this until all the other functions are working. Furthermore, the *powerset* function is specified to be non-destructive to the original set so it must create a new instance and populate it. (Remember that $\mathbb{P}\,S$ contains all the *subsets* of S.)

```
// A set implementation using a clist
// Author: drs

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
#include "clist.h"
#include "set.h"

struct set_implementation
{
    clist *    items;
    printer    item_printer;
    equals     item_compare;
};

set * new_set(printer item_printer, equals item_compare)
{
    set * s = (set *) malloc (sizeof(set));
    assert (s!=NULL);
    s->items = new_clist();
    s->item_printer = item_printer;
    s->item_compare = item_compare;
    return s;
}

int set_isempty(set *s)
{
    assert(s!=NULL);
    return 0; // needs to be implemented
}

int set_isSubset(set *s, set * t)
{
    assert(s!=NULL);
    clist_goto_head(s->items);
    while (clist_cursor_inlist(s->items))
        if (!set_isin(t,clist_get_item(s->items)))
            return 0; // if not in t then not a subset
        else
            clist_goto_next(s->items);
    return set_count(s) < set_count(t); // all in t, but check t is larger than s
}
```

```
int set_isEqualTo(set *s, set * t)
{
    assert(s!=NULL);
    return set_isSubsetEq(s,t) && set_isSubsetEq(t,s);
}

int set_isSubsetEq(set *s, set * t)
{
    assert(s!=NULL);
    clist_goto_head(s->items);
    while (clist_cursor_inlist(s->items))
        if (!set_isin(t,clist_get_item(s->items)))
            return 0; // if not in t then not a subset
        else
            clist_goto_next(s->items);
    return 1;
}

int set_count(set *s)
{
    assert(s!=NULL);
    return clist_size(s->items);
}

int set_isin(set *s, any x)
{
    assert(s!=NULL);
    clist_goto_head(s->items);
    while (clist_cursor_inlist(s->items))
        if (s->item_compare(clist_get_item(s->items),x))
            return 1; // found it
        else
            clist_goto_next(s->items);
    return 0; // not found
}

void set_insertInto(set *s, any x)      // s = s u {x}
{
    assert(s!=NULL);
    if (!set_isin(s,x))
        clist_ins_before(s->items,x);
}

void set_removeFrom(set *s, any x)      // s = s \ {x}
{
    // needs to be implemented
}

void set_intersectWith(set *s, set * t) // s = s n t    t is unchanged
{
    assert(s!=NULL);
```

```
    // needs to be implemented
}

void set_unionWith(set *s, set * t)     // s = s u t    t is unchanged
{
    assert(s!=NULL);
    // needs to be implemented
}

void set_minusWith(set *s, set * t)     // s = s \ t    t is unchanged
{
    assert(s!=NULL);
    // needs to be implemented
}

set* set_powerset(set *s)               // generates new set
{
    assert(s!=NULL);
    return NULL;                        // needs to be implemented
}

void set_print(set *s)
{
    assert(s!=NULL);
    printf("{");
    clist_goto_head(s->items);
    if (clist_cursor_inlist(s->items)) {
        s->item_printer(clist_get_item(s->items));
        clist_goto_next(s->items);
        while (clist_cursor_inlist(s->items)) {
            printf(", ");
            s->item_printer(clist_get_item(s->items));
            clist_goto_next(s->items);
        }
    }
    printf("}");
}

void set_release(set *s)
{
    assert(s!=NULL);
    assert(clist_isempty(s->items));
    clist_release(s->items);
    free(s);
}

int seteq(any s, any t)
{
    return set_isEqualTo((set*)s,(set*)t);
}
```

```
void setprn(any s)
{
    set_print((set*)s);
}
```

Below is an outline test/demo program. You will need to extend this to test the other functions as they become available. We recommend an *incremental development* strategy.

```
#include <stdio.h>
#include "any.h"
#include "set.h"

void intprn(any x)
{
    printf("%li", (long)x);
}

int inteq(any x, any y)
{
    if ((long)x == (long)y)
        return 1;
    else
        return 0;
}

int main()
{
    set * s, *t, *u;

    s = new_set(intprn,inteq);
    t = new_set(intprn,inteq);
    set_insertInto(s,(any)3);
    set_insertInto(s,(any)5);
    set_insertInto(s,(any)7);
    printf("s = ");
    set_print(s);
    printf("\n");
    set_insertInto(t,(any)4);
    set_insertInto(t,(any)5);
    set_insertInto(t,(any)6);
    printf("t = ");
    set_print(t);
    printf("\n");

    u = new_set(setprn,seteq);
    set_insertInto(u,s);
    set_insertInto(u,t);
    printf("u = ");
    set_print(u);
```

```
    printf("\n");

    set_insertInto(s,(any)9);
    printf("s = ");
    set_print(s);
    printf("\n");
    printf("u = ");
    set_print(u);
    printf("\n");
}
```

Practical Exercise 2

A *bag* data structure is a collection in which duplicates are allowed, but the order of the elements is not significant. Thus its protocol lies between a *list* and a *set*. Operations are similar to that of a set except that there is no *intersect* operation; and the *union* and *minus* operations are replaced by *sum* and *difference* operations that take into consideration the number of duplicate values in the collection.

Define a suitable header file and then implement this library using a suitable underlying data structure. Devise a test program to verify the operations in the data structure. As in the case of the *set* data structure, we recommend an incremental development strategy.