

SOFTWARE VERIFICATION RESEARCH CENTRE
DEPARTMENT OF COMPUTER SCIENCE
THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 94-39

**A Haskell implementation
of Z data types**

Mark Utting

December 1994

Phone: +61 7 365 1003

Fax: +61 7 365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

A Haskell implementation of Z data types

Mark Utting
Software Verification Research Centre
University of Queensland
St. Lucia, QLD 4072
AUSTRALIA

Email: marku@cs.uq.edu.au
Phone: +61 7 365 1653

This report describes a Haskell [HPe92] module that implements finite versions of the set, relation and function types from Z [Hay87] [Spi89]. It uses the literate script convention, where each executable Haskell line begins with a > character.

```
> module ZTypes( Set,
>                 set, union, inter, diff, gen_union, gen_inter,
>                 mem, not_mem, subset, psubset, disjoint,
>                 card, set2list, powerset,
>                 Reln(..),
>                 reln, dom, ran, domres, domsub, ranres, ransub,
>                 semi, inverse, image, identity, iter, transclosure,
>                 Func(..),
>                 func, is_func, is_injective, fapply, fdefapply,
>                 fovr, funion, fclash,
>                 -- Sundry useful functions
>                 merge, mergesort, while, remdups, implies, iff
>                 )
> where
>
> infixr 1 'implies'
> infixr 0 'iff'
```

Z expression	Haskell expression
$\mathbb{F} X$	<code>Set X</code>
$\mathbb{P} S$	<code>powerset S</code>
$\mathbb{P}_1 S$	<code>powerset S 'diff' set []</code>
$S = T$	<code>S == T</code>
$S \neq T$	<code>S /= T</code>
$S \in T$	<code>S 'mem' T</code>
$S \notin T$	<code>S 'not_mem' T</code>
\emptyset	<code>set []</code>
$\{a, b, c\}$	<code>set [a,b,c]</code>
$S \subseteq T$	<code>S 'subset' T</code>
$S \subset T$	<code>S 'psubset' T</code>
$S \cup T$	<code>S 'union' T</code>
$S \cap T$	<code>S 'inter' T</code>
$S \setminus T$	<code>S 'diff' T</code>
$\bigcup S$	<code>gen_union S</code>
$\bigcap S$	<code>gen_inter S</code>
<code>disjoint [S1,...,Sn]</code>	<code>disjoint [S1,..,Sn].</code>
$\{x : S \mid P(x) \bullet f(x)\}$	<code>set [f x x <- set2list S, P x]</code>

Table 1: Set expressions in Z and Haskell.

1 Finite Sets: (Set a)

This finite set type is derived from page 228 of Bird and Wadler [BW88]. However, several functions are renamed and many functions are added so that all of the set operations of Z are supported. Table 1 shows the relationship between Z set operators and the corresponding functions defined in this module (S and T stand for arbitrary (finite) set expressions and X stands for a (possibly infinite) Haskell type or its Z equivalent). Note that functions such as `union`, `inter` and `in` are typically used as infix functions (*e.g.*, `a 'mem' xs`).

1.1 Constructor Function

The `set` function constructs a set from a list of elements. The only constraint on the input list is that it must contain a finite number of values whose type is totally ordered (*i.e.*, for any two elements a and b , either $a \leq b$ or $b \leq a$ must be true). In particular, the input list may contain duplicates and does not need to be sorted in any particular order.

```
> set          :: (Ord a) => [a] -> Set a
```

1.2 Set Operations

The three binary operators, `union`, `inter` and `diff` are the same as in Bird and Wadler, except that `differ` has been renamed to `diff`. It is recommended that they be written as infix functions. The `gen_union` and `gen_inter` functions return the distributed union and intersection respectively of a finite set of sets. The empty set is an acceptable argument to `gen_union` (and returns the empty set) but the argument to `gen_inter` must contain at least one set.

```
> union        :: (Ord a) => Set a -> Set a -> Set a
> inter        :: (Ord a) => Set a -> Set a -> Set a
> diff         :: (Ord a) => Set a -> Set a -> Set a
> gen_union    :: (Ord a) => Set (Set a) -> Set a
> gen_inter    :: (Ord a) => Set (Set a) -> Set a
```

1.3 Predicates

Apart from `disjoint`, which takes a sequence (list) of sets, these functions are intended to be used as infix functions. Note that `disjoint [xs,ys,zs]` is equivalent to

```
xs 'inter' ys = set [] &&
xs 'inter' zs = set [] &&
ys 'inter' zs = set [] .
```

```
> mem          :: (Ord a) =>      a -> Set a -> Bool
> not_mem      :: (Ord a) =>      a -> Set a -> Bool
> subset       :: (Ord a) => Set a -> Set a -> Bool
> psubset     :: (Ord a) => Set a -> Set a -> Bool
> disjoint     :: (Ord a) =>      [Set a] -> Bool
```

1.4 Sundry Functions

The `card` function gives the number of elements in a set. The `set2list` function returns a list sorted in ascending order containing the elements of the given set, without any duplicates. The `powerset` function returns the set of all subsets of a given set.

```

> card      :: (Ord a) => Set a -> Int
> set2list  :: (Ord a) => Set a -> [a]
> powerset  :: (Ord a) => Set a -> Set (Set a)

```

The `set2list` function is typically used to simulate set comprehensions using list comprehensions. For example, if `xs` contains a set of integers and `pred` is some predicate over integers, the Z set comprehension

$$\{x \in xs \mid \text{pred}(x) \bullet x^2 - x\}$$

would be written in Haskell as

```
[ x^2 - x | x <- set2list xs, pred x ] .
```

Note that this returns a list of numbers rather than a set, but the list can easily be converted back into a set by using the `set` function, giving

```
set [ x^2 - x | x <- set2list xs, pred x ] .
```

The functions defined so far cover all of the set operations defined in the standard Z library [Spi89], except for some derived operators (such as the non-empty power set constructor: \mathbb{P}_1) that are easily expressed using other functions.

1.5 Implementation

Our implementation of sets uses lists sorted in ascending order with no duplicates. This allows the common operations to have worst case performance of $O(N)$ in the size of the set, except for `set` which is $O(N \times \ln N)$ for unsorted input lists (but $O(N)$ for sorted input lists). If `mem` is by far the most common operation upon sets, balanced binary trees may be a better representation.

Since the implementation assumes that elements can be totally ordered, the element type must be an instance of the `Ord` class. A consequence of this is that sets must also be totally ordered, since we want to allow sets to contain sets. However, the user will generally not be interested in this ordering over sets (it is not the same as the subset ordering), so we regard it as being a side effect of the current implementation rather than as a desired property of sets.

```

> data Ord a => Set a = MkSet [a] deriving (Eq,Ord)

> set xs = (MkSet . remdups . mergesort) xs

```

The implementations of set union, intersection and difference are all variations of a merge operation. We could define union using `remdups` and `merge`, but give a direct definition for symmetry with intersection and difference.

```

> union (MkSet xs) (MkSet ys) = MkSet (union_elems xs ys)
> inter (MkSet xs) (MkSet ys) = MkSet (inter_elems xs ys)
> diff  (MkSet xs) (MkSet ys) = MkSet (diff_elems xs ys)

> union_elems [] ys      = ys
> union_elems (x:xs) []  = x:xs
> union_elems (x:xs) (y:ys) -- merge, removing duplicates
>   | x < y              = x : union_elems xs (y:ys)
>   | x == y             = x : union_elems xs ys
>   | x > y              = y : union_elems (x:xs) ys

> inter_elems [] ys      = []
> inter_elems (x:xs) []  = []
> inter_elems (x:xs) (y:ys)
>   | x < y              = inter_elems xs (y:ys)
>   | x == y             = x : inter_elems xs ys
>   | x > y              = inter_elems (x:xs) ys

> diff_elems [] ys      = []
> diff_elems (x:xs) []  = x:xs
> diff_elems (x:xs) (y:ys)
>   | x < y              = x : diff_elems xs (y:ys)
>   | x == y             = diff_elems xs ys
>   | x > y              = diff_elems (x:xs) ys

> gen_union (MkSet xs)  = foldr union (set []) xs
> gen_inter (MkSet xs)  = foldr1 inter xs

```

Rather than use the standard `elem` function to implement `mem` we give an implementation that takes advantage of the list being sorted. This means that it is only necessary to search half the list on average.

```

> x 'mem' (MkSet xs)    = x 'mem_impl' xs
> x 'not_mem' xs       = not (x 'mem' xs)

> x 'mem_impl' []      = False
> x 'mem_impl' (y:xs)

```

```

> | x > y      = x 'mem_impl' xs
> | x == y     = True
> | x < y      = False

> xs 'subset' ys      = xs 'diff' ys == set []

> xs 'psubset' ys     = xs 'subset' ys && xs /= ys

> disjoint xs
>   = and [ a 'inter' b == set []
>           | (i,a) <- numxs, (j,b) <- numxs, i < j ]
>   where
>     numxs = zip [1..] xs

```

The remaining functions have simple implementations, since our representation of sets does not contain duplicates and is already sorted in ascending order.

```

> card (MkSet xs)      = length xs

> set2list (MkSet xs) = xs

> powerset (MkSet xs) = set (map set (powset xs))
>   where powset []    = [[]]
>         powset (x:xs) = [ x:ys | ys <- powxs ] ++ powxs
>         where powxs = powset xs

```

The following definitions allow sets to be printed and parsed as {a, b, c}. They are taken from the VDM SetMap module by Nick North (email: ndn@seg.npl.co.uk); available by anonymous ftp from ftp.dcs.glasgow.ac.uk. If you don't like the syntax, you can change the brackets and the item separator by changing the following three constants.

```

> smb    = "{ "
> sme    = "}"
> smsep  = ", "
> instance (Ord a, Text a) => Text (Set a) where
>   showsPrec d (MkSet [])      = showString smb . showString sme
>   showsPrec d (MkSet (x:xs)) =
>     showString smb . shows x . foldr g (showString sme) xs
>   where
>     g x' s = showString (smsep ++ " ") . shows x' . s
>

```

```

> readsPrec p =
>   readParen False
>     (\r -> [(set xs, t) | (tok,s) <- lex r, tok == smb,
>                          (xs,t) <- readset s])
>   where
>     readset s = [([],t) | (tok,t) <- lex s, tok == sme] ++
>                   [(x:xs,u) | (x,t) <- reads s, (xs,u) <- readrest t]
>     readrest s = [([],t) | (tok,t) <- lex s, tok == sme] ++
>                   [(x:xs,v) | (tok,t) <- lex s, tok == smsep,
>                               (x,u) <- reads t,
>                               (xs,v) <- readrest u]

```

2 Finite Relations: (Reln a b)

As in Z, we model relations as a set of pairs. Since we want all the set operations to be directly applicable to relations, we define the `Reln` type as a synonym for the type of finite sets that contain pairs. The `reln` function constructs a relation from a list of pairs. It is identical to the set constructor, but is provided to make programs more readable.

```

> type Reln a b = Set (a,b)

> reln      :: (Ord a, Ord b) => [(a,b)] -> Reln a b
> reln ps   = set ps

```

We can now define the following operators on relations. The relationship between the Z operator names and the names used here is shown in table 2. Note that `dom` is short for *domain* and `ran` is short for *range*. Be warned that `iter` only accepts iterations of one or greater, because zero generates the identity relation (see `identity`) and we cannot do this unless we know the complete underlying set of the given relation (and that it is finite).

```

> dom      :: (Ord a, Ord b) => Reln a b -> Set a
> ran      :: (Ord a, Ord b) => Reln a b -> Set b
> domres   :: (Ord a, Ord b) => Set a -> Reln a b -> Reln a b
> domsub   :: (Ord a, Ord b) => Set a -> Reln a b -> Reln a b
> ranres   :: (Ord a, Ord b) => Reln a b -> Set b -> Reln a b
> ransub   :: (Ord a, Ord b) => Reln a b -> Set b -> Reln a b
> semi     :: (Ord a, Ord b, Ord c) =>
>
>           Reln a b -> Reln b c -> Reln a c
> image    :: (Ord a, Ord b) => Reln a b -> Set a -> Set b
> inverse  :: (Ord a, Ord b) =>
>           Reln a b -> Reln b a

```

Z expression	Haskell expression
$X \leftrightarrow X'$	<code>Reln X X'</code>
$\{(a,1), (b,2)\}$	<code>set [(a,1), (b,2)]</code>
$\text{dom } R$	<code>dom R</code>
$\text{ran } R$	<code>ran R</code>
$S \triangleleft R$	<code>S 'domres' R</code>
$S \trianglelefteq R$	<code>S 'domsub' R</code>
$R \triangleright S$	<code>R 'ranres' S</code>
$R \trianglerighteq S$	<code>R 'ransub' S</code>
$R \circledast R'$	<code>R 'semi' R'</code>
$R(S)$	<code>R 'image' S</code>
$R \sim$	<code>inverse R</code>
<code>id</code>	<code>identity S</code>
R^k	<code>R 'iter' k</code>
R^+	<code>transclosure R</code>

Table 2: Relation expressions in Z and Haskell.

```
> identity      :: (Ord a)      =>          Set a      -> Reln a a
> iter          :: (Ord a)      => Reln a a -> Int      -> Reln a a
> transclosure :: (Ord a)      =>          Reln a a -> Reln a a
```

2.1 Implementation

The implementation of `dom` and `ran` is a direct translation of the usual set comprehension definition.

```
> dom rs = set [ x | (x,y) <- set2list rs ]
> ran rs = set [ y | (x,y) <- set2list rs ]
```

The following implementations of domain and range restriction and subtraction have complexity $O(\#rs \times \#s)$, which could be improved considerably by taking advantage of ordering.

```
> domres s rs = set [(x,y) | (x,y) <- set2list rs, x 'mem' s]
> ranres rs s = set [(x,y) | (x,y) <- set2list rs, y 'mem' s]
> domsub s rs = set [(x,y) | (x,y) <- set2list rs, x 'not_mem' s]
> ransub rs s = set [(x,y) | (x,y) <- set2list rs, y 'not_mem' s]
```

The simple implementation of `semi` (called `semi_spec`) has complexity $O(\#xs \times \#ys)$. The more sophisticated implementation that is used for the exported `semi` takes the inverse of the first relation then does a single pass which calculates the cartesian product wherever the domains intersect. The worst case complexity of this implementation is the same as for `semi_spec`, but the average case is much better.

```
> semi_spec (MkSet xs) (MkSet ys)
>   = set [ (x,y') | (x,x') <- xs, (y,y') <- ys, x' == y ]

> semi xs (MkSet ys) = set (match xs' ys)
>   where
>     MkSet xs' = inverse xs
>     match [] _ = []
>     match _ [] = []
>     match xs@((x,x'):xtail) ys@((y,y'):ytail)
>       | x < y = match xtail ys
>       | x > y = match xs ytail
>       | x == y = [(a,b) | a <- x':map snd eqx, b <- y':map snd eqy]
>                 ++ match neqx neqy
>       where
>         (eqx,neqx) = span (\p -> fst p == x) xtail
>         (eqy,neqy) = span (\p -> fst p == y) ytail
```

Disregarding the $O(N \times \ln(N))$ overhead for turning a list into a set, the implementation of `inverse` has complexity $O(\#rs)$, whereas that of `image` is $O(\#rs \times \#s)$. However, `image` could be $O(\#rs + \#s)$ if we used an intersection-like algorithm.

```
> image rs s = set [ y | (x,y) <- set2list rs, x 'mem' s]

> inverse rs = set [(y,x) | (x,y) <- set2list rs]
```

The remaining functions compose a relation with itself various numbers of times (zero for `identity`, a given number for `iter` and one or more for `transclosure`).

```
> identity s      = set [(x,x) | x <- set2list s]

> iter rs 1      = rs
> iter rs (k+1) = rs 'semi' iter rs k

> transclosure rs = grow rs rs
>   where
>     grow orig curr
```

```

>     = if new == set []
>       then curr
>       else grow orig (curr 'union' new)
>       where new = (orig 'semi' curr) 'diff' curr

```

3 Finite Functions: (Func a b)

Functions are a special case of relations, so all the relation and set operators are applicable to functions (though the result may be a relation rather than a function).

Since functions are already available in Haskell, one could ask why it is useful to provide an ADT that represents functions explicitly? The advantage of representing functions as data structures is that it becomes possible to perform operations such as calculating the domain and range of a function, test two functions for equality and print the definition of a function. The disadvantage is that we can only usefully represent finite functions.

The following definitions define `Func` as a synonym for `Reln`, to make programs more readable. The `is_func` predicate tests a relation to determine if it is in fact a function. The `is_injective` predicate tests a relation to see if it is a one-to-one function. The `func` operation constructs a function from a list of pairs—it is identical to `reln`, but generates an error message if the result is not a function.

```

> type Func a b = Reln a b

> is_func      :: (Ord a, Ord b) => Reln a b -> Bool
> is_func rs = card (dom rs) == card rs

> is_injective :: (Ord a, Ord b) => Reln a b -> Bool
> is_injective rs = is_func rs && card (ran rs) == card rs

> func :: (Ord a, Ord b) => [(a,b)] -> Func a b
> func ps
>   = if is_func rs
>     then rs
>     else error "func: list contains duplicate domain elements."
>     where rs = reln ps

```

3.1 Finite Function Operations

Application of a function to a value is performed by the `fapply` function. The given value must be in the domain of the function. Alternatively, if you call `fdefapply` you can supply an additional default value that will be returned if the function argument is not in the domain of the function.

The `fovr` and `funion` operators (usually written infix) combine two functions into one. The difference is that `funion` requires the two argument functions to agree wherever their domains intersect, whereas `fovr` (*function override*) combines any two functions and gives priority to the *second* function whenever a common domain value appears in both inputs. Note that `funion` has exactly the same effect as `union`, except that it raises an error if the resulting relation is not a function.

The `fclash` operator compares two functions and returns the domain elements that appear in both functions but are mapped to different range elements. So, if `fclash` returns an empty set, then `funion` can be safely applied. Although the `funion`, `fovr` and `fclash` operations could be generalised to operate on relations rather than just on functions, the current implementations assume that the inputs are functions.

```
> fapply      :: (Ord a, Ord b) =>      Func a b -> a -> b
> fdefapply   :: (Ord a, Ord b) => b -> Func a b -> a -> b
> fover       :: (Ord a, Ord b) => Func a b -> Func a b -> Func a b
> funion      :: (Ord a, Ord b) => Func a b -> Func a b -> Func a b
> fclash      :: (Ord a, Ord b) => Func a b -> Func a b -> Set a
```

3.2 Implementation

```
> fapply rs v
>   = fdefapply (error "fapply: argument is not in domain of function")
>       rs v

> fdefapply def (MkSet xs) v
>   = f def xs v
>     where
>       f def [] v
>         = def
>       f def ((x,x'):xs) v
>         | v < x  = def
>         | v == x = x'
>         | v > x  = f def xs v
```

The implementations of `fivr` and `funion` are very similar to the implementation of set union. The implementation of `fclash` is similar to that of set intersection. All three operations have complexity of $O(\#xs + \#ys)$.

```
> fivr (MkSet xs) (MkSet ys) = set (fivr' xs ys)
>   where
>     fivr' [] ys           = ys
>     fivr' (x:xs) []      = x:xs
>     fivr' ((x,x'):xs) ((y,y'):ys)
>       | x < y            = (x,x') : fivr' xs ((y,y'):ys)
>       | x == y           = (y,y') : fivr' xs ys
>       | x > y            = (y,y') : fivr' ((x,x'):xs) ys

> funion (MkSet xs) (MkSet ys) = set (funion' xs ys)
>   where
>     funion' [] ys@(_:_ ) = ys
>     funion' xs@(_:_ ) [] = xs
>     funion' ((x,x'):xs) ((y,y'):ys)
>       | x < y            = (x,x') : funion' xs ((y,y'):ys)
>       | x == y && x' == y' = (x,x') : funion' xs ys
>       | x > y            = (y,y') : funion' ((x,x'):xs) ys
>       | otherwise        = error "funion: functions clash"

> fclash (MkSet xs) (MkSet ys) = set (clash xs ys)
>   where
>     clash [] ys           = []
>     clash (x:xs) []      = []
>     clash ((x,x'):xs) ((y,y'):ys)
>       | x < y            = clash xs ((y,y'):ys)
>       | x == y && x' == y' = clash xs ys
>       | x == y && x' /= y' = x : clash xs ys
>       | x > y            = clash ((x,x'):xs) ys
```

4 Sundry Useful Functions

4.1 Mergesort

Sorts a list into ascending order with $O(N \times \ln N)$ worst case time, but with $O(N)$ best case performance when the list is already ordered. The elements in the list must come from a type which is totally ordered.

```

> mergesort :: Ord a => [a] -> [a]
> mergesort xs
>   = (head . while gt1 mergepairs . ascend_runs) xs
>     where
>       gt1 [_] = False
>       gt1 _   = True

> merge []      ys      = ys
> merge xs@(_:_ ) []    = xs
> merge (x:xs) (y:ys)
>   | x < y      = x : merge xs (y:ys)
>   | x == y     = x : merge xs (y:ys)
>   | x > y      = y : merge (x:xs) ys

> while :: (a -> Bool) -> (a -> a) -> a -> a
> while tst f xs
>   = if tst xs
>     then while tst f (f xs)
>     else xs

> mergepairs :: Ord a => [[a]] -> [[a]]
> mergepairs []          = []
> mergepairs [a]         = [a]
> mergepairs (a:b:xs)    = merge a b : mergepairs xs

> ascend_runs :: Ord a => [a] -> [[a]]
> ascend_runs []         = [[]]
> ascend_runs (x:xs)     = asc_runs x xs
>   where
>     asc_runs v []       = [[v]]
>     asc_runs v (x:xs)
>   | v <= x              = (v:ys) : zs
>   | v > x               = [v] : asc_runs x xs
>     where ys:zs = asc_runs x xs

```

4.2 Remdups

Removes adjacent duplicates from a list. A corollary of this is that if the input list is sorted (in ascending or descending order) then the output list will only contain unique values.

```

> remdups :: (Eq a) => [a] -> [a]
> remdups []           = []
> remdups [a]         = [a]
> remdups (a:b:as)
>   | a == b          = remdups (b:as)
>   | otherwise       = a : remdups (b:as)

```

4.3 Logical Operators

For ease of translating Z predicates to Haskell, we define two Z -like logical connectives, `implies` and `iff`. Note that the Z \wedge , \vee and \neg operators can be translated to the standard Haskell operators `&&`, `||` and `not`, respectively. Of course, the semantics of the Haskell operators is slightly more restricted than the Z operators, since the Haskell operators typically evaluate the left argument before the right, so they may be undefined (i.e., not terminate) in some cases where the Z operators are defined.

```

> implies :: Bool -> Bool -> Bool
> implies a b = not a || b
>
> iff :: Bool -> Bool -> Bool
> iff a b = a == b

```

References

- [BW88] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, 1988.
- [Hay87] Ian J. Hayes. *Specification Case Studies*. Prentice-Hall, 1987.
- [HPe92] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.
- [Spi89] J. Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice-Hall, 1989. ISBN 013983768X.