Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 201-218

EFFICIENT IMPLEMENTATION OF LINEARLY USED FINITE MAPS

P. Diviánszky (Budapest, Hungary)

Abstract. Typical functional implementations of finite maps are based on association lists or balanced binary trees. This paper describes a nontypical implementation using mutable references internally.
The interface of the data structure is similar to the interface of the `Data.Map` module in Haskell. The main difference is that keys of maps can only be *identifiers*. Identifiers are members of an abstract data type. The implementation provides functions which helps the creation and distribution of identifiers.
If the number of used maps has an upper bound and the maps are used linearly, then the insert, delete and search operations are *constant time* operations independently of the number of keys. A number of pointer algorithms can be implemented with finite maps so that these constraints hold. These algorithms could also be implemented with mutable references, but the use of finite maps gives a more functional look and feel. Prototype implementation of the data structure with usage examples exists in Clean and Haskell. This paper describes the main points of the implementations and presents two usage examples: an algorithm which finds strongly connected components in directed graphs and a sequence data structure implementation.

1. Introduction

Let us begin with a puzzle. Suppose that we have expressions `apply`, `f`, `g`, `x`, `y` and `z` in Haskell so that the following equations hold:

```
apply f x == 15
apply f y == -2
apply f z == 0
apply g x == "abcd"
apply g y == "14x"
```

What are the definitions of these expressions? For example, `f` and `g` can be finite maps. The concrete representation may vary so that the type of `apply` is

`apply :: (a → b) → a → b,`	in case of functions,
`apply :: Eq a ⇒ [(a, b)] → a → b,`	in case of association lists,
`apply :: Ord a ⇒ Map a b → a → b,`	in case of size-balanced trees.

(The `Data.Map` module in Haskell implements the `Map` data structure with size-balanced binary trees.)

Let us see a non-standard solution to the puzzle. Let us call `f` and `g` *maps* and `x`, `y`, `z` *keys*. In the previous solution the maps hold the values like 15, -2, "`abcd`". Now change this so the keys contain the values. Let `f` and `g` be arbitrary expressions and assume that `f` ≠ `g`. The definitions of `x`, `y`, `z` and `apply` are

```
x = [(f, 15), (g, "abcd")]
y = [(f, -2), (g, "14x")]
z = [(f, 0)]
apply a l = case lookup a l of
    Just b → b
    Nothing → error "..."
```

The `lookup` function searches its first argument in the second argument, which is an association list in this case. The code will not work in Haskell because of typing errors; we will see a solution for that later. In which case can be the last solution to the puzzle superior to the previous ones? Suppose that there are lots of keys `x`, `y`, `z`, ..., and only a few maps. In this case it is worth to interchange the roles of keys and maps because every key will contain at most as many entries as the number of maps, so `apply` will run in constant time independently of the number of keys.

This paper shows a finite map data structure implementation with the run-time behaviour of the last solution to the puzzle. This data structure is more efficient than the existing finite map implementations in certain cases. As we will see these are the cases when the maps are used linearly, and the number of maps are much smaller than the number of keys.

2. Module interface of the maps

In a typical implementation of finite maps the type of keys can be any data type which has ordering (`Ord` instance in Haskell). In our case there is more stress on keys because they will also contain the values in a hidden way. This stress forces us to restrict the type of keys to be an abstract data type. The abstract data type could hold arbitrary values, too, but for simplicity I chose identifiers as keys. Identifiers are members of a very simple abstract data structure: the only operation on identifiers is testing their equality. There are also functions to construct identifiers, of course.

2.1. Type indexed identifiers

Identifiers are members of the abstract data type `Id` which is defined in Haskell as follows:

```
data Id k
```

The type `Id` is indexed by the type `k`. The interface of identifiers ensures that `k` can only be an uninstantiated type variable: one can not create a valid (non-bottom) identifier of type `Id Int`, for example. The type index makes possible the distinction between identifiers by the type system. For example, we can implement a type-safe bipartite graph with type-indexed identifiers. A bipartite graph is a special graph where the set of vertices can be divided into two disjoint sets U and V such that no edge has both end-points in the same set. We can model the two sets with types `Id a` and `Id b`. The two sets are disjoint if `a` $\neq$ `b` because `a` and `b` can not be instantiated by a common type. The graph contains list of edges from `Id a` to `Id b` and list of edges from `Id b` to `Id a`:

```
data BipartiteGraph a b = Graph [Edge (Id a) (Id b)] [Edge (Id b) (Id a)]
type Edge a b = (a, b)
```

The only thing that can be done with identifiers is comparing of identifiers with identical index. There is an equality instance for identifiers:

```
instance Eq (Id k)
```

2.2. Creation of identifiers

Suppose that we can produce a value of type T provided that we have a list of different identifiers [`Id k`]. In other words, we have a function `f` :: [`Id k`] $\to T$.

Suppose moreover, that T does not contain the type variable `k`. This means that identifiers indexed by `k` can not "leak out" from the expression of type T, so it is safe to construct this expression with arbitrary identifiers which are pairwise different. In other words, one can safely create new identifiers at any point of the program, provided that these identifiers will not interact with identifiers created at an other point of the program.

Fortunately, we can get the Haskell type system with rank-2 types to check that newly created identifiers will not "leak out":

```
runIds :: (∀ k . [Id k] → a) → a
```

The inner ∀ guarantees that the type variable `a` can not be instantiated by a type which depends on `k`. The same method is used in [6].

Recursive functions consuming identifiers can thread through a list of fresh identifiers, but then some aspects of functional programming will be lost. Instead, in these cases we can use the following function:

```
split :: Id k → [Id k]
```

The `split` function generates new identifiers. The new identifiers are fresh unless they are generated from the same identifier.

2.3. Interface of the maps

The interface of the proposed maps is similar to the interface of the `Map` data type in the Haskell `Data.Map` module which implements size balanced binary trees based finite maps [1]:

`data Map k a`

> A map from keys `Id k` to values `a`. Note the difference between `Map k a` and `Map (Id k) a`. The second would mean a map from keys `Id (Id k)` to values `a`. Because identifiers can only be indexed by an uninstantiated type variable, the second map has no sense.

`empty :: Map k a`

> The empty map.

`insert :: Id k → a → Map k a → Map k a`

> Insert a new key and value in the map. If the key is already present in the map, the associated value is replaced with the newly supplied value.

`delete` :: `Id k` → `Map k a` → `Map k a`

Delete a key and value from the map. If the key is not present in the map, the result is the unchanged input map.

`lookup` :: `Id k` → `Map k a` → `Maybe a`

Lookup the value at a key in the map.

A specialised version of `lookup` is the (!) operator:

```
(!) :: Map k a → Id k → a
m ! i = fromJust (lookup i m)
```

Note that it is not possible to obtain the list of keys of a map defined here. Lots of operation relay on the list of keys: traversals, conversions, filtering, submaps, indexing, operations with minimum and maximum elements. We have to miss all these features. On the other hand, it is possible to define an other finite map data structure on top this one which caches the keys of the map, and provides the previous operations. Eventually the absence of traversal, etc. operations can be seen as an advantage: in pointer algorithms we do not need them (see later), so we can use a more efficient finite map implementation.

3. Implementation

The implementation should cope with the following tasks:

- Identification of keys and maps;
- Keys should store mutable values;
- Keys should store values hidden from the user;
- Proper handling of non-linear use.

The used techniques are:

- The key identification, and the storing of mutable values are solved with mutable variables from the `Control.Concurrent.MVar` module in the base package of GHC.

- Maps are identified by integers. Fresh integers are drown form a global variable implemented by a mutable variable and use of the `unsafePerformIO` function.
- The hidden store of values is solved by abstract data types and Dynamics (`Data.Dynamic` module in the base package).
- The proper handling of non-linear use is solved by a patching technique inspired by the implementation of diff arrays in the `Data.Array.Diff` module in the base package of GHC.

3.1. Basics of patching

First let us look at simplified versions of the data types used in the implementation. On this simplified version we can see the previously mentioned patching mechanism.

The definition of the `Id` type:

```
newtype Id k = Id (AssocList MapId Dynamic)
type AssocList a b = [(a, b)]
```

Identifiers contain (`MapId`, `Dynamic`) pairs. Dynamics are needed because the key holds values with different types if the key is present in different maps.

The index `k` is not present in the left-hand-side of the definition, because the index is needed only during the type checking of the users code.

Each map has a unique map identifier which is an integer value in the implementation:

```
type MapId = Int
```

The Map type:

```
data Map k a
     = Pure MapId
     | Patched (Id k) (Maybe a) (Map k a)
```

The map is pure, if all its data corresponding keys and values are stored in the keys. If the map is patched, then it is a slightly modified version of an other map. The modification can be either the deletion of a key from the map, or an update of a value at a key, or an insertion of a key and value pair. In either case, the key is stored as the first parameter of the patch. In case of deletion, the second parameter is `Nothing`. There is no distinction between insertion and update. In both cases, the new value is wrapped in a `Just` constructor. According this here is the simplified definition of lookup:

```
lookup :: Id k → Map k a → Maybe a
lookup id m = case m of
    Pure mapId →
        case lookup mapId id of
            Nothing → Nothing
            Just d → fromDynamic d

    Patched id′ val m′
        | id′ == id → val
        | otherwise → lookup id m′
```

3.2. Current and old maps

To understand the implementation of insert and delete, let us define the notions *current* and *old* maps. During the run of the program the lifecycle of a map is: current → old. If an insert or delete is performed on a current map, it will be old; the newly created maps (by insertion or deletion) are current. Note that in case of linear use every insert and delete will be performed on a current map. Note that the evaluation order depends on the evaluation strategy. The examples in this paper are defined so that maps are used linearly in case of *a lazy* evaluation strategy (Haskell and Clean have lazy evaluation strategy).

3.3. The patch-transformation

The simplest solution is to implement `insert` and `delete` is that we add a patch:

```
insert′ id a m = Patched id (Just a) m
delete′ id a m
    | isNothing (lookup id a m) = m
    | otherwise = Patched id Nothing m
```

This is not optimal; our goal is to ensure fast run in case of linear use. The best would be the case when all current maps are pure. This can be achieved by the *patch-transformation* steps.

Here I describe one of the three patch transformation steps; the other two are similar. Suppose that `a` and `b` are maps and `id` is an identifier, so that `a` is pure with map identifier `mapId`. Suppose that `b` is a patched form of `a` where the patch is an update `a` at `id` from the old value `x` to the new value `y`:

```
a = Pure mapId
b = Patched id (Just y) a
id = (mapId, x): other
```

Now we can define the maps a′ and b′ and the identifier id′ so that replacing a, b, id with a′, b′, id′ does not alter the semantics of maps (the expressions (a, b, id) and (a′, b′, id′) are contextually equivalent from the view of the user):

```
a′ = Patched id (Just x) b′
b′ = Pure mapId
id′ = (mapId, y): other
```

With the three patch transformations it is always possible that the outcome of a delete or insert on a pure map will also be pure.

3.4. Final data types

The patch transformations should exchange all maps envolved at once (atomically). In Haskell this can be implemented with mutable variables combined with `unsafePerformIO`.

Every time when a Haskell programmer uses an unsafe language feature, a justification is needed that its use is safe in that particular case. In our case `unsafePerformIO` is used in the implementation of the patch-transformations, which are safe because they do not alter the semantics of expressions. The final data types are:

```
newtype Id k = Id (MVar (AssocList MapId Dynamic))
type AssocList a b = [(a, b)]
type MapId = Int

data Map k a

     = Map (MVar (MapData k a))
     | Empty

data MapData k a
     = Pure MapId
     | Patched (Id k) (Maybe a) (Map k a)
```

Identifiers and maps are changed by the patch transformation, so these are wrapped in mutable variables. The `Empty` constructor of maps is needed only for technical reasons (see later).

3.5. Some other function definitions

Let us see the implementation of some of the functions on identifiers and maps.

The implementation of `runIds` is rather simple; we create new mutable variables and apply the input on them:

```
runIds :: (∀ k . [Id k] → a) → a
runIds f = f [unsafePerformIO (newMVar []) | _<-[1..]]
```

Note that the GHC compiler may optimise this code so that all identifiers will be the same mutable variable in the list. This can be prevented with additional tricks which are not presented here.

For the creation of a new map we need a unique map identifier (of type `Int`). The solution is a global mutable variable which holds an integer. Global mutable variables can be implemented with `newMVar` and `unsafePerformIO`:

```
numOfMaps :: MVar Int
numOfMaps = unsafePerformIO (newMVar 0)
```

When a new empty map is created, the counter is incremented and its value will be the identifier of the map. The naive implementation of `empty` would be `empty = unsafePerformIO emptyIO`. In this case all empty maps would share the same map identifier, which is wrong. The workaround is that we introduce a new `Empty` constructor and write additional function alternatives for the `Empty` case:

```
empty :: Map k a
empty = Empty

insert id x Empty = insert id x (unsafePerformIO emptyIO)
insert id x (Map m) = ...
```

4. Finite sets

Finite sets can be naturally implemented on top of finite maps. `Set k` contains identifiers indexed with `k`. `Set k` is a finite map from identifiers indexed with `k` to the unit value:

```
newtype Set k = Map.Map k ()
```

We use qualified names with `Map` prefix to distinguish between function of maps and sets. The implementations of the set functions are very simple:

```
empty :: Set k
empty = Map.empty

insert :: Id k → Set k → Set k
insert k s = Map.insert k () s member :: Id k → Set k → Bool
member k s = isJust (Map.lookup k s)
```

From now on, set operations will be used qualified with the `Set` prefix and map operations will be used without qualification.

5. Usage example: Strongly connected components

The graph representation used in this section is a function from graph nodes to list of graph nodes. Graph nodes are represented by identifiers, so

```
type Graph k = Id k → [Id k].
```

5.1. Graph walks

The graph walk algorithm has two parameters: a graph and a list of initial graph nodes. The result is the list of nodes which are reachable from the initial nodes, in depth-first order.

```
walk :: Graph k → [Id k] → [Id k]
walk children l = collect Set.empty l where

    collect :: Set k → [Id k] → [Id k]
    collect s [] = []
    collect s (h:t)
        | Set.member h s = collect s t
        | otherwise = h : collect (Set. insert h s) (children h ++ t)
```

The `walk` function uses a local function called `collect`. The parameters of `collect` is the set of already visited nodes and the list of nodes to be visited. Seemingly the visited nodes are stored in the set. What happens during execution is that the set itself stores no information; the information that the node is in the set is stored in the graph node. So the `walk` function uses the `Set` data structure to *mark* visited nodes. That is exactly like an imperative graph walk algorithm works.

With some modification of the previous algorithm we get the reachable nodes in postorder (post-ordering of the depth-first forest [5], suitable for strongly connected components detection. We need the `Task` auxiliary data structure to mark reached nodes:

```
data Task k = Return (Id k) | Visit (Id k)

walk′ :: (Id k →[Id k]) → [Id k] → [Id k]
walk′ children l = collect Set.empty (map Visit l) where

    collect :: Set k → [Task k] → [Id k]
    collect s [] = []
    collect s (Return h: t) = h: collect s t
    collect s (Visit h: t)
```

```
    | Set.member h s = collect s t
    | otherwise

        = collect (Set.insert h s) (map Visit (children h) ++ Return
        h: t)
```

The `collect` local function gets the set of visited nodes and a list of tasks. This list contains values which are either nodes to visit or nodes to return. If the first element in the list should be returned, it is returned. If the first element should be visited, and it was not already visited, then the following tasks are added to the list of tasks: visit the children of the node and return the node.

5.2. Relation inversion

Suppose that the relation between elements of type `a` and `b` is represented as pair of an `a` → [`b`] function and a domain of type [`a`]. The `inverse` function inverts a relation between elements of type `a` and identifiers index by `b`:

```
inverse :: (a→[Id b]) → [a] → (Id b→[a])
inverse r domain = get r' where


    r' = foldl add empty [(x, y) | x←domain, y ←r x]


    add :: Map b [a] → (a, Id b) → Map b [a]
    add m (a, b) = l `seq` insert b (a: l) m where l = get m b
    get :: Map b [a] → Id b → [a]
    get m x = case lookup x m of

        Just l → l
        Nothing → []
```

First the relations are transformed into pairs: [(`x`, `y`) | `x`←`domain`, `y`←`r x`]. Then an empty map from `Id b` to [`a`] is created which will hold the result relation. The pairs are added to the map one-by-one. At the end, the map is transformed to a `Id b`→[`a`] function by the `get` function. Note that in function `add`, the ‘`seq`‘ function ensures that `l` = `get m b` is evaluated before `insert b` (`a`: `l`) `m`. Without it, `l` would be evaluated after the insertion into `m`, so a lookup would happen (in the `get` function) after an insertion into `m`, so the use of `m` would be non-linear. This would not be a problem, but the algorithm would be slower.

5.3. Strongly connected components

The strongly connected components can be obtained by a reversed postorder walk and a walk with reversed children relation [5].

The result is a tuple of the number of components and a map which maps a node to the number of component which contains the node.

```
scc :: (Id a→[Id a]) → [Id a] → [[Id a]]
scc children l = walk'' (inverse children l') l' where

    l' = reverse (walk' children l)

walk'' :: (Id a→[Id a]) → [Id a] → [[Id a]]
walk'' children l = snd $ mapAccumL (collect []) Set.empty (map (:[]) l) where
    collect :: [Id a] → Set a → [Id a] → (Set a, [Id a])
    collect acc s [] = (s, acc)
    collect acc s (h:t)
        | Set.member h s = collect acc s t
        | otherwise = collect (h: acc) (Set.insert h s) (children h ++ t)
```

The `walk''` function maps a lists of nodes into list of children of that nodes. If a node is once collected, it will not show up again as a children. The `Set` data structure is used to remember which node was already collected.

The time and space consumption of the component search algorithm (and the other algorithms presented here) is $O(n)$ where n is the number of nodes, provided that the graphs have relatively few edges per nodes.

6. Usage example: Implementation of the Sequence data structure

In the base package of GHC the Data.Sequence module contains the implementation of the `Seq a` data type. `Seq a` is a double ended list of elements of type `a`, implemented by finger trees [3]. One can add and delete elements in $O(1)$ time on its both ends. Here is the implementation of `Seq a` with finite maps. The data structure is defined as:

```
data Seq a
    = Empty
    | ∀ k . Seq
        { value :: Map k a
        , first, last :: Id k
```

```
    , prev, next :: Map k (Id k)
    , ids :: [Id k]
    }
```

A sequence is either an empty sequence or an existentially quantified record. (Existential quantification is denoted by the ∀ keyword in Haskell.) With existential quantification the `k` type index can be omitted from the type of the sequence, so the interface of `Seq` will be the same as the one defined in `Data.Sequence`.

The spine of the sequence consists of identifiers indexed by `k`. The identifiers are mapped to values by the `value` map. `first` and `last` are the first and last identifiers of the spine, respectively. The `prev` and `next` maps determine the order of identifiers in the spine. Two invariants of `prev` and `next` are:

`next ! (prev ! i) == i` if `i` $\neq$ `first`,
`prev ! (next ! i) == i` if `i` $\neq$ `last`.

(There are other invariants observed by the operations, too.) The `ids` field contains fresh identifiers. Fresh identifiers are needed when new elements are added to the sequence.

To make a singleton list one have to use the `runIds` function.

```
singleton :: a → Seq a
singleton a = runIds f where
    f (id:ids) = Seq
        { value = insert id a empty
        , first = id
        , last = id
        , prev = empty
        , next = empty
        , ids = ids
        }
```

Insertion to the left end of the sequence:

```
(<|) :: a → Seq a → Seq a
a <| Empty = singleton a
a <| (Seq value first last prev next (first′:ids)) = Seq
    { value = insert first′ a value
    , first = first′
    , last = last
    , prev = insert first first′ prev
    , next = insert first′ first next
    , ids = ids
    }
```

Other operations have similar complexity in code. The efficient implementation of the `length` operation needs a cached length field in the record above. The constant time implementation of the (`><`) infix operation which join two sequences is future work. This sequence data structure has similar time and space behaviour to the sequence in Data.Sequence, but only in case of linear use. So the tradeoff is between simpler implementation and more liberate use.

Other data structures, like double ended lists with a moving pointer, rings, or any classical pointer-based data structure can be implemented with similar code and time complexity.

7. Implementation in Clean

The implementation of the presented map in Clean has both advantages and drawbacks.

7.1. Advantages with Clean

The advantages come from the graph rewriting semantics behind Clean and the uniqueness type system in Clean [2]. The uniqueness type system is used in two ways:

- Initial identifiers are unique and splitting is permitted only on unique identifiers:

  ```
  runIds :: (A. k: *[Id k] → a) → a
  split :: *Id k → *[*Id k]
  ```

  The effect is that some misuse of `split` will be rejected by the compiler (splitting the same identifier twice) and the implementation of `split` will shorter and faster (no need to remember the splitted values).

- The association list of identifiers can be updated destructively, because they are never shared. In the Clean implementation this is achieved by a variant of `unsafeCoerce#` which alters the uniqueness attribute of the association list. The Clean compiler generates code which do "compilation time garbage collection" on unique objects.

The graph rewriting semantics behind Clean has also an effect on the language. One effect is that there is a distinction between constants and zero arity function definitions. Constants are computed only once, while zero arity functions

are computed every time when used. This distinction helps the implementation at two points: the definition of the empty map and the definition of the fresh identifier (which is an internal impure function). Both can be defined as zero arity functions and thus the complexity of the implementation reduces a bit. (For example, the `Empty` constructor defined in Sect. 3.4 is not needed.)

7.2. Drawbacks with Clean

The drawbacks are that there are no weak pointers and mutable variables implementation in Clean.

The implementation of mutable variables is possible without changing the compiler. One can use ABC code for this purpose. The code can be derived from the `Heap` module in the sources of the Clean compiler frontend, which implements the `Heap` and `Pointer` data structures.

Implementing weak pointers in Clean would be a huge work so I had to miss them.

8. Current problems

The main problem with the current implementation is its wrong memory usage. The identifiers hold the values binded to them by maps in association lists. Currently the unused elements in the association lists are not garbage collected. The problem can be solved by weak pointers [4] in Haskell: the garbage collection of an unused map can trigger the garbage collection of the related items in the association lists of identifiers. However, when I tried this solution I observed decrease in performance. Another problem, that there are no weak pointers in Clean.

The consequences are increased space consumption and possible space leaks. The proposed solution is described in the next section.

9. Future work

9.1. Static check of linear use

The linear use of maps could be checked by the type system. Unfortunately, the uniqueness type system of Clean can not do this. The `insert` and `delete` functions could have type

```
insert :: Id k → a → *Map k a → *Map k a
delete :: Id k → *Map k a → *Map k a
```

(* denotes uniqueness). On the other hand, after one use of `lookup` one could not use the unique map any more because of the uniqueness restrictions. A solution would be to modify `lookup` so that it returns the same map:

```
lookup′ :: Id k → *Map k a → (a, *Map k a)
```

I disagree with this solution because the maps should be threaded in the code, which makes the code less "functional".

Ensuring linear use of the presented maps is a motivating challenge to improve the Clean uniqueness type system.

9.2. Compaction of the association lists of identifiers

In some cases, the association lists of identifiers could be replaced by a record data structure.

Consider the sequence implementation in Sect. 6. Almost every identifier in the spine of the sequence has an internal association list which consists of values of the maps `value`, `prev` and `next`. (The first and last identifier in the spine contains only two of them.) It would be nice if the compiler could replace these list with a record data type like

```
Id_Seq a = Id_Seq
     { value :: a
     , prev :: Id_Seq a
     , next :: Id_Seq a
     }
```

The optimised `insert`, `delete` and `lookup` functions would operate on these records. The resulted data type would be very similar to a double-linked list implemented in *C* with structures and pointers.

This optimisation step seems feasible, but it needs first the static check of linear use of maps.

9.3. Improved memory use

The wrong memory use of the maps should be cured. Currently the only solution I see is the modification of the garbage collector in the run time environment to treat the association lists of identifiers specially.

10. Conclusion

The finite map implementation described in this paper is suitable to implement pointer algorithms in a functional way. The resulted code is also efficient. However, a problem with memory consumption is not solved yet. Until this problem is solved, the greatest contribution of this paper is to show how can one implement a pointer algorithm with finite maps.

References

[1] **Adams S.,** Efficient sets: a balancing act, *J. Functional Programming,* **3** (4) (1993), 553-562.

[2] **Barendsen E. and Smetsers S.,** Uniqueness typing for functional languages with graph rewriting semantics, *Mathematical Structures in Computer Science,* **6** (6) (1996), 579-612.

[3] **Hinze R. and Paterson R.,** Finger trees: a simple general-purpose data structure, *J. Functional Programming,* **16** (2) (2006), 197-217.

[4] **Jones S.L.P., Marlow S. and Elliot C.,** Stretching the storage manager: weak pointers and stable names in haskell, *Implementation of Functional Languages,* 1999, 37-58.

[5] **King D.J. and Launchbury J.,** Structuring depth-first search algorithms in haskell, *Conf. Record of POPL'95: 22nd Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, San Francisco, CA,* ACM Press, New York, 1995, 344-354.

[6] **Launchbury J. and Jones S.L.P.,** Lazy functional state threads, *PLDI'94: Proc. ACM SIGPLAN 1994 Conf. on Programming Language Design and Implementation,* ACM Press, New York, 1994, 24-35.

P. Diviánszky
Department of Programming Languages and Compilers
Eötvös Loránd University
Pázmány Péter sét. 1/C
H-1117 Budapest, Hungary
divip@aszt.inf.elte.hu