Lisp and Symbolic Computation, 2, 105–113 (1989)

Arrays in Pure Functional Programming Languages

KLAUS E. GRUE
DIKU, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen East, Denmark (grue @ diku. dk)

Abstract. In pure functional programs it is common to represent arrays by association lists. Association lists have the disadvantage that the access time varies linearly both with the size of the array (counted in number of entries) and with the size of the index (counted in cons nodes). This paper presents another simple representation of arrays for which the access time varies linearly in the size of the index but is independent of the size of the array. The paper compares this representation with association lists in functional languages and arrays in imperative languages.

This paper also considers lazy programming and states how to use potentially infinite arrays for time optimization for certain programs.

1. Introduction

Lisp [9, 10] has infinitely many constructors: **cons**, and one for each atom. Hence, the domain of Lisp, i.e. the set of S-expressions, is infinitely generated. Contrarily, the domains for HyperLisp [11] and Formal Language [5] are finitely generated. The generators are {**cons**, **snoc**, **nil**}, and {**cons**, **nil**}, respectively. Furthermore, most implementations of Lisp are finitely generated. As an example, the domain of an implementation of Lisp that offers a function for concatenation of atom names is generated by this concatenation function, the atoms whose names consist of one character, and **cons**.

The representation of arrays presented in this paper works for finitely generated domains, i.e., array indices must be restricted to some finitely generated domain. For languages with finitely generated domains, array indices may vary over the entire domain of the language. Hence, one need not impose restrictions on array indices in HyperLisp, Formal Language, and most implementations of Lisp. The range of values that may be stored in arrays never needs to be restricted.

The representation of arrays works best for finitely generated domains with few constructors, and becomes particularly simple for the domain of Formal Language which is generated by {**cons**, **nil**}. This paper states how to represent arrays for this domain and describes how to treat other cases.

As practical application of languages with few constructors seems quite new [5, 11], and as their use is not widespread, we now list some advantages of such languages.

1. It is possible to make languages with few constructors which have simple and

mathematically clean semantics compared to other ones. This makes them easier to learn, implement, and prove properties about. In particular, structural induction is simpler for such languages.

2. Languages with few constructors are in line with the RISC (Reduced Instruction Set Computer) philosophy, which claims that a simple computer architecture can be made more efficient than a complicated one, because it is easier to optimize something simple than something complex. In particular, finitely generated languages may be executed on reduction machines [2, 14] which have few reduction rules. As an example, Formal Language can be implemented using 9 reduction rules [5]. The SK–system [1, 3, 14] merely requires 2 reduction rules, but this reduction system is very inefficient, and the semantics is complicated due to the lack of an axiom of induction in the SK–system.
3. Simple, finitely generated languages are suited to reflection and meta-reasoning. As an example, Formal Language was made for the following purpose: A proof checker for proving properties of Formal Language programs was expressed in Formal Language and used for proving properties about itself. The aim of doing so is described in [5]. Proofs about a quick and dirty proof checker are much longer than proofs about a simple and clean one, so even if the quick and dirty proof checker was faster than the simple and clean one counted in proof lines per hour, it would still be slower for the above, reflective task. The above task requires a simple, clean language and proof checker. The proof checker actually used proved 40,000 short proof lines per hour.

The representation of arrays has been tried out in Formal Language. Formal Language has been in use since 1986 at the University of Copenhagen. Apart from proof construction and verification, it has been used for compiler writing and simulation of imperative computers.

For the convenience of the reader, the algorithms are stated in Lisp in this paper. We use the style of [7] for writing Lisp algorithms. We concentrate on domains generated by **cons** and **nil** and state how to generalize the results.

Other approaches to efficient representation of arrays suited for purely functional programming languages may be found in [8, 12, 13]. The representation in [8] aims at providing arrays as a built in facility in new programming languages whereas the representation in the present paper allows arrays to be expressed in existing languages. The representation in this paper has some resemblance with balanced trees [12, 13]. However, the representation in the present paper differs from balanced trees in that indexing requires traversal of a path that merely depends on the index and in that access times are predictable.

2. Representation by finite trees

As it might not be obvious that languages with as few constructors as **cons** and **nil** are as expressive as other ones, possible representations of integers and Lisp S–expressions are given below.

Let **F** be the domain generated by **cons** and **nil**, i.e. **F** contains all Lisp S–expressions in which no other atoms than **nil** occur. We refer to the elements of **F** as 'finite trees'. We now represent integers and Lisp S–expressions by finite trees.

One may represent binary 0 and 1 as follows:

0 is an abbreviation for **nil**.

1 is an abbreviation for (**nil. nil**).

The binary number 10100 may be represented by the list

(1 0 1 0 0)

Let f be an enumeration of the atoms of Lisp, i.e., f assigns a unique, positive integer $f(a)$ to each atom a of Lisp. Let $g(a)$ be the integer $f(a)$ represented in **F** as above. For each S–expression s we may now assign a finite tree $h(s)$, where h is defined as follows:

$h(a) =$ **cons** $g(a)$ **nil** for atoms a

$h($**cons** $s\ t) =$ **cons** $h(s)$ **cons** $h(t)$ **nil** for S–expressions s and t.

As any S–expression can be represented in **F**, any Lisp program can. At this point it should be obvious that one can do anything in languages with no other constructors than **cons** and **nil** that can be done in other ones. In the above representation, a **cons** in Lisp is represented by two **cons** in Formal Language, and the number of nodes used for representing integers is linear in the logarithm of the integer, which is to live with.

3. Finite maps

In what follows, we express programs in Lisp in the notation of [7]. As an example consider the following definition of *'append'*:

append $x\ y =$ **if** $x \equiv$ **nil then** y **else cons car** x *append* **cdr** $x\ y$.

With this definition we have

append (**nil nil**) ((**nil nil**) **nil**) = (**nil nil** (**nil nil**) **nil**).

For convenience, define

head $x =$ **if** $x \equiv$ **nil then nil else car** x.

tail $x =$ **if** $x \equiv$ **nil then nil else cdr** x.

We call $f: \mathbf{F} \to \mathbf{F}$ a 'finite map' if $f(x)$ is defined for all $x \in \mathbf{F}$ and if $f(x) \neq \mathbf{nil}$ merely holds for finitely many x. Finite maps correspond to association lists in functional and arrays in imperative languages. Let **A** be the set of finite maps.

In order to represent finite maps, we shall define two functions *'index'* and *'override'*, and one constant (i.e., finite tree) *'empty'* such that:

1. $\forall f \in \mathbf{A} \exists t \in \mathbf{F} \forall x \in \mathbf{F}$: $\quad index\ x\ t = f(x)$.

2. $\forall x \in \mathbf{F}$: $\quad index\ x\ empty = \mathbf{nil}$.

3. $\forall x, y, t \in \mathbf{F}$: $\quad index\ x\ override\ x\ y\ t = y$.

4. $\forall x, y, z, t \in \mathbf{F}$: $x \neq z \Rightarrow$ $\quad index\ x\ override\ z\ y\ t = index\ x\ t$.

We say that $t \in \mathbf{F}$ represents $f \in \mathbf{A}$ if $\forall x \in \mathbf{F}$: $index\ x\ t = f(x)$. Hence, (1) expresses that all finite maps should be representable, (2) expresses that *'empty'* should represent the finite map f for which $\forall x \in \mathbf{F}: f(x) = \mathbf{nil}$, and (3) and (4) express that if t represents f and if g is given by

$$g(z) = y$$

$$g(x) = f(x) \qquad \text{for } x \neq z,\ x \in \mathbf{F}.$$

then (*override z y t*) should represent g.

Condition (1) follows from (2), (3), and (4).

4. The function '*index*'

The function '*index*' defined later may be computed as follows: let $x, t \in \mathbf{F}$. Any x may be expressed using a combination of **nil** and **cons**. Assume as an example that x is

cons nil cons nil nil.

To compute (*index* (**cons nil cons nil nil**) t), proceed as follows. First, textually replace **nil** by **head** and **cons** by **tail** as in

tail head tail head head.

Next, write this backwards as in

head head tail head tail.

Last, apply this to t:

index **cons nil cons nil nil** $t =$

head head tail head tail t.

The computation proceeds likewise for all x. The definition of '*index*' is easy to state using the above:

index x $t =$ **if** $z \equiv$ **nil then head** t **else** *index* **tail** x *index* **head** x **tail** t.

Define

empty = **nil**.

Having **head nil** = **nil** and **tail nil** = **nil**, it is obvious that '*index*' and '*empty*' satisfy (2).

Example: Let $A, B, C \in \mathbf{F}$, and define f by

f(**nil**) = A.

f(**cons nil nil**) = B.

f(**cons nil cons nil nil**) = C.

$\forall x \in \mathbf{F} \setminus \{$**nil, cons nil nil, cons nil cons nil nil**$\}$: $f(x)$ = **nil**.

The function f is a finite map. The finite tree in Figure 1 represents it:

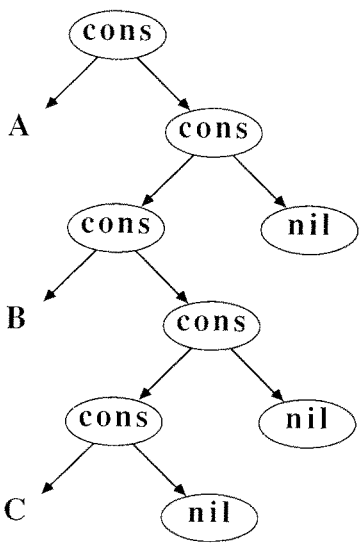

Fig. 1.

We now explain why '*index*' satisfies (1). For each x, $(index\ x\ t)$ is defined as a head-tail sequence of functions applied to t. Hence, for each x, $(index\ x\ t)$ addresses some location in the tree t. To obtain $(index\ x\ t) = y$ for some $x, y \in \mathbf{F}$, we merely have to insert y at the proper location in t. In Figure 1, A, B, and C are inserted at their proper locations.

In Figure 1, A, B, and C are independent in the sense that we may change one of them without affecting the others. This is not true for all locations. As an example, we cannot change **head tail tail head** t without affecting **tail head** t. In general, if **X** and **Y** are head-tail sequences, and if one of them is a suffix of the other, then one cannot change **X** t without changing **Y** t.

Let $u, v \in \mathbf{F}$. Computation of $(index\ u\ t)$ and $(index\ v\ t)$ requires some head-tail sequences **U** and **V**, respectively, to be applied to t. From the definition of **U** and **V** as transliterated polish postfix forms it follows that if $u \neq v$, then neither of **U** and **V** are suffixes of the other. Hence, (1) holds.

5. The function '*override*'

The function '*override*' requires some auxiliary definitions:

semicons $x\ y =$ **if** $x \equiv$ **nil** $\wedge\ y \equiv$ **nil then nil else cons** $x\ y$.
notnil $=$ **cons nil nil**.
Polishprefix $x\ y =$ **if** $x \equiv$ **nil then cons nil** y **else**
cons *notnil Polishprefix* **head** x *Polishprefix* **tail** $x\ y$.
override1 $x\ y\ t =$ **if** $x \equiv$ **nil then** y **else**
if (**head** x) $\equiv$ **nil**
then *semicons* (*override1* **tail** $x\ y$ **head** t) (**tail** t)
else *semicons* (**head** t) (*override1* **tail** $x\ y$ **tail** t).
override $x\ y\ t =$ *override1* (*Polishprefix* x **nil**) $y\ t$

The function '*override*' satisfies (3) and (4). The use of '*semicons*' makes '*override*' attempt to minimize the returned finite tree.

6. Other domains than F

It is straightforward to modify *empty*, *index*, and *override* to work with domains generated by other constructors than **cons** and **nil**. Another possibility seems more convenient: Suppose we want arrays whose indices are restricted to some set **G** where $\mathbf{G} \neq \mathbf{F}$. All we need to do is to program a function $g{:}\mathbf{G} \to \mathbf{F}$ such that $\forall x, y \in \mathbf{G}{:}(x \neq y \Rightarrow g(x) \neq g(y))$. We may then define

empty–**G** $=$ *empty*

$$index\text{–}\mathbf{G}\,x\,t \quad = index\ (g\,x)\ t$$

$$override\text{–}\mathbf{G}\,x\,y\,t = override\ (g\,x)\ y\,t$$

In general, the penalty in memory and execution time consumption of the above approach is to live with. We now return to considering the domain **F**.

7. Comparison with arrays in imperative languages

Arrays in imperative languages require a fixed amount of memory whereas the finite maps described above expand and shrink dynamically as needed. Array bounds have to be declared before using arrays whereas users of finite maps have free hands. Sparse arrays make inefficient use of memory whereas finite maps automatically reward sparseness.

If an imperative program needs two almost identical arrays, then the two arrays together require twice as much memory as one array. Two almost identical finite maps may share memory, such that they together only require a little more memory than one of them.

Array access requires a fixed, small amount of time. Finite map access requires more time, and the time required is linear in the size of the index (counted in nodes). We have represented integers by finite trees such that the size of a tree is proportional to the logarithm of the size of the integer. Hence, for finite maps indexed by integers, the access time is proportional to the logarithm of the index.

Normally, arrays can be indexed by integers and enumerated types only. Finite maps may be indexed by integers, records, other finite maps, algorithms, and everything else that may be represented by finite trees.

Indexing by integers and algorithms has been tried in practice, and indexing by finite maps has been considered for the implementation of a language built upon finite maps rather than S–expressions. Indexing by algorithms has been used in a proof checker in which, at any time, a finite number of algorithms were classified as valid proof procedures.

8. Comparison with association lists in functional languages

As mentioned in the abstract, the access time for association lists depends linearly on the size of the array whereas the access time for finite maps is independent of the size. Hence, finite maps are obviously better than association lists for large arrays.

To compare two finite trees x and y, which happen to be equal, it is necessary to inspect all nodes of both x and y. To index an association list, it is necessary to compare the index to each entry in the list. If the index happens to be among the entries, then the index is eventually compared to an entry equal to the index, which

requires each node of both index and entry to be inspected. To index a finite map, it is merely necessary to inspect each node of the index, which is twice as fast as comparing the index to the entry. Hence, finite maps are strictly better than association lists with respect to run time performance whenever the index is among the entries. It is possible to express the function '*index*' such that finite maps are better than or as good as association lists in all cases.

9. Memory consumption

Let a and b be finite trees with M and N **cons** nodes, respectively. Let $t = override\ a\ b\ empty$. The finite tree t represents the map f for which $f(a) = b$ and $f(x) =$ **nil** for $x \neq b$. The finite tree t contains exactly $2M + N + 1$ **cons** nodes. Of these $2M + N + 1$ nodes, N are used for representing b and $2M + 1$ are used for the path in t to b.

Let $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ be finite trees with $M_1, \ldots, M_n$ and $N_1, \ldots, N_n$ **cons** nodes, respectively. Let $t = override\ a_1\ b_1\ override\ a_2 b_2 \ldots override\ a_n b_n\ empty$. The finite tree t has at most $2M_1 + \ldots + 2M_n + N_1 + \ldots + N_n + n$ **cons** nodes. The precise size of t depends on $a_1, \ldots, a_n$ and the amount of memory sharing. For comparison, the association list corresponding to t has at most $M_1 + \ldots + M_n + N_1 + \ldots + N_n + n$ **cons** nodes.

10. Potentially infinite arrays

As defined, $f: \mathbf{F} \to \mathbf{F}$ is a finite map if $f(x)$ is defined for all $x \in \mathbf{F}$ and if $f(x) \neq$ **nil** merely holds for finitely many x. As shown above, finite maps can be represented by finite trees. Lazy languages [4, 6, 7] like Formal Language [5] allow S-expressions to be infinitely large as long as merely finite parts of the S-expressions actually have to be computed and kept in memory. Such potentially infinite S-expressions are capable of representing any map $f: \mathbf{F} \to \mathbf{F}$, i.e. $f(x) \neq$ **nil** may now hold for infinitely many x. The functions *index* and *override* stated earlier works for such potentially infinite maps.

Example: let f be a potentially infinite map which satisfies (say)

$$index\ n f = n^{10}$$

for all integers n (where integers are represented as before). Such an f is easy to define in any lazy language. Each time $(index\ n f)$ is calculated for some n, one of two things happens: (1) If $(index\ n f)$ has not been calculated before for that n, then n^{10} is calculated, the value of n^{10} is inserted at address n in the internal representation of f, and the value of n^{10} is returned. (2) If $(index\ n f)$ has been calculated before, then the value of n^{10} is extracted from f rather than calculated again. Hence, the benefit of using $(index\ n f)$ in place of n^{10} is that n^{10} is calculated once only for each value of n. The cost is that the value of n^{10} for each new n has to be kept in memory. This results in a traditional time/memory trade off between using $(index\ n f)$ and calculating n^{10}

directly. It is worth noticing that, using a lazy language, the computer does all the work on updating *f* without being told explicitly by the programmer. The programmer merely has to define *f* and use *index*, then the computer takes care of the rest.

11. Conclusion

Compared to imperative arrays, finite maps have slower access times, but finite maps have so many advantages, that they are realistic competitors.

Compared to association lists, finite maps have short access times, and the access times are independent of the size of the array.

Acknowledgments

My thanks are due to Nils Andersen, DIKU, for comments on the manuscript.

References

1. Barendregt, H. P. The lambda calculus, its syntax and semantics. *Studies in Logic and The Foundations of Mathematics 103*, North-Holland 1984.
2. Berkling, K. J. *Reduction languages for reduction machines*. ISF-76-8, Gesellschaft für Mathematik und Datenverarbeitung MBH Bonn, Institut für Informationssystemforschung, 1976. (Also shorter version in *2nd Annual Symposium Computer Architecture*, Houston, 1975).
3. Burge, W.H. *Recursive programming techniques*. Addison-Wesley, 1975.
4. Friedman, D. P., Wise, D. S. CONS should not evaluate its arguments. 257–284 in: Michaelson, S. and Milner, R., Ed., *Automata, Languages and Programming*. Edinburgh University Press, 1976.
5. Grue, K. E. An efficient formal theory. Diku report 14/87, Diku 1987.
6. Henderson, P. and Morris, J. H., Jr. A lazy evaluator. *Conference Record of the Third ACM Symposium on Principles of Programming Languages*, ACM, 1976.
7. Henderson, P. *Functional programming, application and implementation*. Prentice-Hall, 1980.
8. Hughes, J. *An Efficient Implementation of Purely Functional Arrays. To appear.*
9. McCarthy, J. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM*, 1960, 184–195.
10. McCarthy, J., Abrahams, P. W., Edwards, D. J., Hart, T. P. and Levin, M. I. *LISP 1.5 Programmer's Manual*, The M.I.T. Press, 1962.
11. Sato, M. Theory of symbolic expressions, I. *Theoretical Computer Science* **22**, (1983), 19–55.
12. Tarjan, R. E. Data Structures and Network Algorithms, *CBMS-NSF Regional Conference Series in Applied Mathematics* **44**, SIAM (1983).
13. Tarjan, R. E. Algorithm Design, *Communications of the ACM*, **30** 204–212, (March 1987).
14. Turner, D. A. A New Implementation Technique for Applicative Languages. *Software Practice and Experience* **9** (1979) 31–49.