ソフトウェア基礎論 37－3
プログラミング言語 27－3
(1990. 12. 13)

Arrays in Miranda

讃多美 マーティン

Martin Santavy

Department of Computer Science
and Communication Engineering
Kyushu University

九州大学工学部情報工学科

Abstract. We discuss a functional approach to manipulating arrays in parallel. We use a set of primitives that are efficiently implementable in a parallel environment, and show that, when combined with higher-order operators, even a very rudimentary set of primitives can be used to express algorithms in a clean, elegant fashion that is inherently parallel. Miranda provides us with the power and elegance of a functional programming environment and allows us to effectively combine the basic operations into more complicated functions.

Introduction

This paper discusses an algebra of objects called arrays. Our arrays are similar to matrices in mathematics and arrays in Pascal or FORTRAN. Similarly to matrices, they can be manipulated by a number of methods and they conform to a set of laws. Similarly to arrays of high-level programming languages, they are closely related to the physical structure of a computer.

It is not the purpose of this text to develop a full-scale system to manipulate arrays. Rather, we will describe a minimal set of simple, yet powerful operations that allow us to manipulate large segments of data. These operations are designed to utilize the power of parallel environments in a way that does not depend on features of any particular architecture. At the same time, they aim to be flexible, powerful, and "clean" enough to be useful theoretical tools, that can be part of the design of parallel algorithms and reasoning about them.

Let us point out a few, perhaps a little subjective, observations. Firstly, in parallel environments, we want to avoid references to individual data. When a problem or the algorithm to solve that problem is described in terms of individual data items, it always require an effort to convert this "sequential" description to an algorithm that accesses and processes data in parallel. An extreme example would be an algorithm in which individual data show an unbreakable linear dependency, such as in the well-known example of a non-parallelizable problem, to calculate $a^{2^k}, k = 1, 2 \ldots$. One reason why this problem is easy on a sequential machine and hard on a parallel machine lays in the method of describing the problem: it deals with individual pieces of the sequence. Without any reference to the index k it would be difficult to come up with such an example. When indices are used, it is easy for a sequential machine to calculate all pieces of data in a simple loop, and it is irrelevant for the time complexity of the algorithm whether or not the iterations of the loop are independent of each other. On a parallel machine, on the other hand, it is crucial that the "iterations" of a parallel loop are independent and thus parallelizable. In our approach, we want to avoid

the problem of parallelizing loops altogether. We hope to develop an alternative way of describing a (restricted) class of problems that would be inherently parallel, as much as indices and loops are inherently sequential.

Secondly, we admire the power, flexibility, and clarity of a sequential, step-by-step language. Instead of trying to modify the structure of a programming language to allow the programmer to execute parts of her program in parallel, we parallelize the data and leave the language to manipulate them sequential. The efficiency of the programs will lie in the efficiency of the basic tools that programmer uses to manipulate the data. These tools must be designed in a way that permits an efficient implementation in most parallel environments. We try to keep the number of basic tools to the minimum and build more tools on top of this basic set. The efficiency of the more advanced tools is then based on the efficiency of the "primitives", the basic tools.

Thirdly, we would like to use a modern, functional--language approach that would allow us to reason about the algorithms, prove laws and properties, and verify our programs. In this paper we use Miranda. Miranda is a sequential language that does not allow anything but a simulation of the operations on arrays that we would like to see to be executed fast and in parallel. However, it offers a polymorphic, strongly-typed, functional language environment with lazy evaluation, abstract data types, and currying. We use this functional power to combine our primitives in a simple and elegant way. In the design of primitives themselves we settle for a theoretical consideration of possible parallel implementations, and implement them in Miranda in an inefficient, sequential, but demonstrative way.

Array Definition

We will try to avoid the exact definition of an array. There are many ways how to view an array, and selecting one of them as our only definition would be counterproductive.

An array is a multidimensional, rectangular structure that can be fully described by its shape and values. To describe the shape, we use a finite vector of natural numbers. The numbers indicate the size of the array along the corressponding dimensions; the length of the shape vector is the dimensionality of the array. The values of the array form a vector of length equal to the product of the shape. All values are of the same type, and in this text they will be reals.

Alternatively, a k-dimensional array can be viewed as a k-dimensional rectangular grid with a unique value associated with each gridpoint. The advantage of this view is that all axes (dimensions) are equivalent and it is natural to associate a k-dimensional index with each value/gridpoint. However, in a computer representation we have to "flatten" the grid and convert it to some other representation.

A popular view of arrays is influenced by LISP. An array is viewed as a list of lists, with each sublist representing a subarray of the array. The level of nesting of the array/list is its dimensionality. The influence of LISP is reflected in the inclusion of functions "first", "rest", and "cat" (compare "CAR", "CDR", and "LIST" in LISP) in our system. The powerful function "mu" is nothing but LISP's "MAPCAR".

Most of all, of course, we are interested in arrays as abstract data structures to hold actual data. In this sense, we will consider the array to be a set of data items that can be manipulated by our set of primitives with the defined results.

Representation of Arrays

In a computer, arrays must store values. That usually happens in the passive part of the machine, the memory, even though, e.g., on a Connection Machine there is no clear distinction between the active and the passive parts of the computer.

When a shared-memory machine is used, the array can be stored in a continuous block of memory in the row-major order. Multiple processors can then independently manipulate different parts of the array, with the

limited bandwidth of the bus being partially offset by memory caches or small amounts of local memory for each processor. Operations that access data in whole blocks will be efficient. We have to be careful with operations in which each processor must access many different parts of the array, as in a transposition or in a rotation of the array.

When a machine with no shared memory is used, the array must be distributed among the local memories of individual processors. This can be done either by using the combination of the processor address and the local offset in the same way the shared memory address was used to store the array in the row-major order (except that each row is now distributed among several processors) or by using some form of a communication scheme among processors, e.g. a tree or a mesh. This choice, naturally, depends on the physical characteristic of a given system. It is no longer possible to view the array as easily-accessible continuous blocks of data. However, the choice of efficient operation remains almost the same: it is still efficient to let each processor work on its "own" piece of the array as much as possible. The communication cost of a logical addressing scheme (such as a tree or a mesh) tend to be higher than when using the physical addresses of the processors to index the array. When a logical addressing scheme is used, however, it is much easier to change the structure of the array.

We try to avoid unnecessary changes in the structure of the array. In array expressions, arrays are combined, and new, both temporary and permanent arrays can be created. Allocation of free memory and its availability to different processors could pose a serious problem whose solution depends on the architecture of the machine. Another difficult problem concerns the method of storage of sparse arrays. We do not address these issues. Instead, we hope for the best and simply try to design the instructions in a way that manipulates very large pieces of data at once while it does not assume any particular architecture or feature of a parallel machine.

We assume MIMD and allow different operations to be used on different chunks of data. The "flattenning" of the array into the row-major order is given by the mapping function $\gamma_s i = \sum_{k\geq 0} i_k \prod_{l>k} s_l$, where $(i_k)_{k\geq 0}$ and $(s_k)_{k\geq 0}$ are the index and the shape, respectively. Its inverse function is defined as $\gamma'_s a = ((a \bmod \prod_{l\geq k} s_l) \operatorname{div} \prod_{l>k} s_l)_{k\geq 0}$, where a is the address. In Miranda, functions $\gamma_s i$ =`gm i s` and $\gamma'_s a$ =`gm' a s` are defined as follows:

```
gm :: [num]->[num]->num
gm i s
= foldl g 0 (zip2 (i++(repeat 0)) s)
  where g r (a,b)=a+(b*r)

gm' :: num->[num]->[num]
gm' a s
= map2 (mod)(foldr g [a](tl s)) s
  where g n (x:xs)=x div n:x:xs
```

Array Representation in Miranda

An array is represented by its shape and values. In Miranda, we use an array constructor, **A**, to construct a pair of a shape vector and a value vector, both of type **[num]**. The type of the array is then called **reparray**. Its synonym, **array**, is used as a name of the abstract data type that includes the array representation, **reparray**, and a set of basic function to manipulate it, **first, rest, cat, isempty**, etc.:

```
reparray ::= A [num] [num]
array == reparray
absttype array
with
  first, rest :: array->array
  cat         :: array->array->array
  isempty     :: array->bool
  ....
```

This set of functions include the printing functions. We print the values of scalars (0-dimensional arrays) as they are. The values of a vector (1-dimensional arrays) are printed as a sequence of values on a line surrounded by angle brackets. Any other array is printed as a sequence of its subarrays separated by new-lines and surrounded by square brackets, except for an empty array, for which only its shape is printed.

A new array can be created using function **arr ss vs**, which returns an array of shape **ss** with whose values taken from vector **vs**. If necessary, vector **vs** is repeated.

As Miranda is a lazy-evaluated language, the values of an array are not accessed unless they are needed. Thus it is possible to create arrays whose values form infinite lists or do not exist at all. In fact, this feature of the language very well suits our needs: when all we need is the shape of an array, its values should be undefined.

In this text we describe a minimal set of primitives that allows us to manipulate arrays in a non-trivial way.

Basic Operations-Primitives

The Miranda definitions given here and in the next chapter define the behaviour of the functions. They do not reflect their implementation on an actual parallel machine. It is not always clear which functions should be primitives ("internal") and which functions should be defined using the primitives. It is possible that a function, originally defined using the primitives, will prove to be so useful that it will be worth to be implemented as a primitive.

The basic set of operations must include functions that give information about the structure of the array. Although this may seem trivial, in fact it is not: e.g. when the shape information is needed several times during the computation, we must must either keep a copy of it at each node of the distributed system or retrieve it and re-broadcast it every time it is requested. The decision which method to use is a tradeoff, whose resolution depends, again, on the concrete characteristics of the system. Examples of structural-information functions are the function **shp x** that returns a list of values representing the shape of the vector **x** and the function **issingle x** that returns the boolean value **True** if and only if the array **x**, when viewed as a list of subarrays, has only one element.

```
shp (A ss vs) = ss
issingle (A (s:ss) vs) = s=1
```

The next come functions influenced by LISP: **first**, **rest**, **cat**, returning the first subarray, everything but the first subarray, and a catenation of two arrays of the same (or a similar) shape along the 0-th dimension, respectively. Since they are simple and they keep contiguous blocks of arrays intact, they are easy to be executed in parallel. The functions follow a simple set of rules. An example of a such a rule is

```
cat (first x) (rest x) == x
```

which holds for for any x. The functions are defined as

```
first (A (s:ss) vs)
= A ss (take (product ss) vs)

rest (A (s+1:ss) vs)
= A (s:ss) (drop (product ss) vs)

cat (A (s1:ss) vs1) (A (s2:ss) vs2)
= A (s1+s2:ss) (vs1++vs2)
cat (A ss vs1) (A (s2:ss) vs2)
= A (1+s2:ss)  (vs1++vs2)
cat (A (s1:ss) vs1) (A ss vs2)
= A (s1+1:ss)  (vs1++vs2)
```

From FP and Miranda we have functions **tk** and **dp**, that treat arrays as lists of lists and behave similarly to Miranda's functions **take** and **drop**. The only difference is that when a negative argument is used, the indicated number of elements is taken or dropped from the end of the array-list. In addition to these two functions, we define function **bk** that returns a specified number of subarrays, starting from a specified position. Since arrays are multidimensional, the functions accept a list of values indicating the number of elements to be taken or dropped in each dimension. Both **tk** and **dp** use a higher-order operation **mu1**, which is defined later. It is worth noticing that

```
cat (tk [n] x) (dp [n] x) == x
```

for any non-scalar **x** and non-negative **n**. For other types of arguments the equation does not hold. Equation

```
bk a b == (tk a).(dp b)
```

Holds by definition. Therefore,

```
tk a == bk a []
dp a == bk [] a
```

The definitions of **dp**, **tk**, and **bk** are as follows:

```
dp [] a = a
dp [s1] (A (s:ss) vs)
= tk [s+s1] (A (s:ss) vs), s1<0
= A (s-s1:ss)
  (drop (s1*(product ss)) vs), 0<=s1<s
= A (0:ss) [], s<=s1
dp (s1:ss1) a = mu1 (dp ss1) (dp [s1] a)

tk [] a = a
tk [s1] (A (s:ss) vs)
= dp [s+s1] (A (s:ss) vs), s1<0
= A (s1:ss)
  (take (s1*(product ss)) vs), 0<=s1<s
= A (s:ss) vs, s<=s1
tk (s1:ss1) a = mu1 (tk ss1) (tk [s1] a)

bk ss1 ss2 a = tk ss1 (dp ss2 a)
```

Using these primitives, we can construct other functions, e.g. a function `cnst c x` that returns the array `x` with all values replaced by `c`:

```
cnst n x = arr (shp x) (repeat n)
```

Functionals and Higher-Order Operations

A functional accepts other functions as its arguments to produce a value. A higher-order operation, on the other hand, produces other operations as its result. In Miranda, we often define it as a functional that accepts multiple arguments. When some of the arguments are omitted, the functional, due to the currying, becomes a higher-order operation.

An example of a simple functional is the reduction, `fldr`.

```
fldr f a
= first a, issingle a
= f (first a) (fldr f (rest a)), otherwise
```

Even though here it is defined recursively, as a primitive it should be defined as a fan-in that runs in logarithmic time on architectures that allow "non-local" communication. The result is then defined only for associative functions.

LISP's function MAPCAR applies its function-argument to all elements of its list-argument. Similarly, the higher-order operation `mu1 f x` applies function `f` to every subarray of array `x`, which is viewed as a list of its subarrays. On parallel machines, the same function is applied to all elements of the array. The structure of the array changes, however. When the structural information is held locally, it is easy to modify it. When it is held in a global location with each processor just holding its address, it has to be duplicated and modified for every subarray. The addresses of these duplicates must then be passed to the processors. Function `mu1` is defined as:

```
mu1 f x
= arr (0:shp (f (first x))) [], hd (shp x)=0
= cat (f (first x))(mu1 f (rest x)),otherwise
```

We introduce two simple generalizations of `mu1`. The operator `mu1'` accepts an additional numerical parameter that determines along which axes the function-argument should be applied. The operator `mu1''` applies its function-argument on the subarrays along the 0th dimension, but supplies the function also with the ordinal number of each subarray, starting from zero for the first subarray:

```
mu1'(n+1) f  = mu1 (mu1' n f)
mu1'  0   f  = f

mu1'' f x
= g n x where n = hd(shp x)
  g (k+1) x = cat (f (n-(k+1))(first x))
                  (g k (rest x))
  g 0 x = arr (0:shp (f 0 (first x))) []
```

There are other possible generalizations of the `mu1` operator, e.g. for binary function-arguments. We will not discuss these generalizations in this text, however.

The higher-order operations `op1` and `op2` extend common unary and binary operations to arrays. When the sizes of the arrays do not match, `op2` tries to replicate the "smaller" argument to match the size of the other argument. E.g., when a scalar is added to a multidimensional array, its value is replicated and added to every element of the array. Operation `mu1` is used to achieve this effect. On any parallel machine the implementation of these operations should be trivial. Except a possible broadcast of the smaller argument in the case of a binary

function there is no communication required. All operation can be done locally and all processors do exactly the same computation.

```
op1 f (A ss vs)
= A ss (map f vs)

op2 f (A s1 v1) (A s2 v2)
= A s1 (map2 f v1 v2), s1=s2
= mu1 (op2 f (A s1 v1)) (A s2 v2), #s1<#s2
= mu1 (op2 g (A s2 v2)) (A s1 v1), #s1>#s2
  where g a b = f b a
```

The access operations `tk`, `dp`, and `bk` are destructive: even though they allow us to access and modify a part of the array, we lose the other parts of the array in the process. The higher-order operation `upd a b f` (update) restricts the function `f` to the part selected by `bk a b`, while leaving the rest of the array intact. Its implementation in a parallel environment should be similar to that of the `bk` function.

```
upd :: [num]->[num]->
       (array->array)->array->array
upd ss1 ss2 f x
= g 1 ss1 ss2 (shp x) f x where
  g (k+1) (s1:ss1) (s2:ss2) (s:ss) f x
  = mcat a (mcat b c), s1>=0 & s2>=0
  = mcat b (mcat c a), s1>=0 & s2<0
  = mcat a (mcat c b), s1<0 & s2>=0
  = mcat c (mcat b a), s1<0 & s2<0
    where
    mcat = mu2' k cat
    a = mu1' k (tk [s2]) x
    b = g (k+2) ss1 ss2 ss f
        (mu1' k (bk [s1] [s2]) x)
    c = mu1' k ((dp [s1]).(dp [s2])) x
  g (k+1) [] (s2:ss2) (s:ss)
  = g (k+1) [s] (s2:ss2) (s:ss)
  g (k+1) [] [] ss f = f
  g (k+1) (s1:ss1) [] = g (k+1) (s1:ss1) [0]
```

The `it` (iterate) operator applies a given function recursively on smaller and smaller parts of a given array. It is not a primitive.

```
it ss1 ss2 f x
= x, isempty x
= upd ss1 ss2 (it ss1 ss2 f) (f x), otherwise
```

LU Decomposition Algorithm

Let us consider the following Pascal routine to decompose matrix `A` into a product `L*U` of a lower-diagonal matrix `L` and an upper-diagonal matrix `U`. The diagonal of `L` is formed of zeros. At the end of the routine, the lower-diagonal part of matrix `A` is replaced by the lower-diagonal part of matrix `L` (excluding the diagonal) and the upper-diagonal part of matrix `A` is replaced by the upper-diagonal part of matrix `U` (including the diagonal).

```
procedure LU (n:integer; var A:matrix)
  { n denotes the size of the matrix A  }
  { type matrix = array[0..,0..]of real }
var i,j,k:integer;
begin
   for k=0 to n-1 do
   begin
      for i:=k+1 to n-1 do {step 1}
         a[i,k]:=a[i,k]/a[k,k];
      for i:=k+1 to n-1 do {step 2}
         for j:=k+1 to n-1 do
            a[i,j]:=a[i,j]-a[i,k]*a[k,j];
   end
end;
```

It is not difficult to verify that this algorithm indeed finds a LU decomposition of a given matrix. Let us simply ignore the problem of division by zero when `a[kk]`=0 here.

The loops in the Pascal routine do not represent the real idea of the algorithm. The real idea is to subsequently take smaller and smaller submatrices of matrix A, and for each submatrix divide its first column, except the upper-left-corner element, by that upper-left-corner element, and then to subtract the outer product of this column with the first row, again, without the upper-left-corner element, from the lower right principal minor (i.e. what is left after removing the first row and column) of the submatrix.

In our system, the the upper-left-corner element of a matrix `x` is accessed as `first(first x)`, while the first column, except the first element, as `bk [n-1,1] [1,0] x where n=hd(shp x)`. Therefore, the update of the submatrix in step 1 can be written as:

```
step1 x
= upd [n-1,1] [1,1]($divd (first(first x)))x
  where n = hd(shp x)
```

The outer-product is very easy to defined when the operator `mu1'` is used.

```
op x y = mu1' k (mult x) y where k=dim y
```

where `mult` is defined, predictably, as `op2(*)`. The first column and the first row, both without their first element, of a matrix `x` can be accessed by `mu1 first(rest x)` and `rest(first x))`, respectively. The lower right principal minor is simply `dp [1,1]`, or `bk [] [1,1]`. Therefore, the step 2 of the algorithm can be written as

```
step2 x
= upd [] [1,1] ($sub
  (op(mu1 first(rest x)) (rest(first x)))) x
```

The consecutive application of steps 1 and 2 is then simply

```
lu = it [] [1,1] (step2.step1)
```

To verify the correctness of the result, and also to taste more of the practical value of the update operator, `upd`, we create two functions to extract the lower and upper triangular portions of the resulting matrix, and a function `chklu` that prints the expected and actual results of the algorithm. Since the function `ip` that calculates the inner product of two arrays, needs yet-undefined operator, `mu2`, we will leave the question of its definition open.

```
lower
= mu1'' g where
  g k=(upd [1][k](cnst 1)).
      (upd [][k+1](cnst 0))

upper = mu1'' g where
        g k=(upd [k] [] (cnst 0))

chklu x
= [lu_x, ip (lower lu_x)(upper lu_x), x]
   where lu_x= lu x
```

Conclusion

The Pascal routine in the example contains a triply-nested loop. Our Miranda equivalent contains none. An argument can be made that the `ip` operator in fact represents a loop because of the tail-recursion in its definition. The index, k, however is gone from the algorithm, and so are the "parallelizable" inner loops and their indices. We see two main positive sides of our approach:

- when the primitives are implemented in parallel, the whole algorithm is inherently parallel, and
- when the algorithm is free of unnecessary loops and indices, it is much easier to see its basic ideas and reason about it.

In most of the existing functional languages, the emphasis lies on lists and their sequential manipulation. The program-verification schemes are based on lists and their properties. Our approach, on the other hand, manipulates whole arrays at once. We hope that this could be useful for the formal verification of nontrivial matrix and array algorithms.

In the future, it will be necessary to show that our approach is flexible enough to be used as a practical tool for designing algorithms. It is still not clear to us what is the optimal set of primitives to be used, especially when sparse matrices are considered.

Acknowledgements

I thank Lenore Mullin and Nathan Freedman for their support and encouragement.

Examples

Let us assume the following definitions:

```
a1 = arr [3..5] [1..]
a2 = tk [2,3] a1
a3 = bk [2,1,3] [0,2,1] a1
a4 = tk [1] a1
```

```
a5 = mu1 (tk [1]) a1
a6 = mu1 (mu1 (tk [1])) a1
a7 = upd [2,1,3] [0,2,1] (op1(*10)) a1
```

Then the results look as following:

```
REQUEST: a1
ANSWER:
shape: <3 4 5>
[[
  <1 2 3 4 5>
  <6 7 8 9 10>
  <11 12 13 14 15>
  <16 17 18 19 20>
][
  <21 22 23 24 25>
  <26 27 28 29 30>
  <31 32 33 34 35>
  <36 37 38 39 40>
][
  <41 42 43 44 45>
  <46 47 48 49 50>
  <51 52 53 54 55>
  <56 57 58 59 60>
]]
```

```
REQUEST: a2
ANSWER:
shape: <2 3 5>
[[
  <1 2 3 4 5>
  <6 7 8 9 10>
  <11 12 13 14 15>
][
  <21 22 23 24 25>
  <26 27 28 29 30>
  <31 32 33 34 35>
]]
```

```
REQUEST: a3
ANSWER:
shape: <2 1 3>
[[
  <12 13 14>
][
  <32 33 34>
]]
```

```
||=======================================
REQUEST: a4
ANSWER:
shape: <1 4 5>
[[
  <1 2 3 4 5>
  <6 7 8 9 10>
  <11 12 13 14 15>
  <16 17 18 19 20>
]]
```

```
REQUEST: a6
ANSWER:
shape: <3 4 1>
[[
  <1>
  <6>
  <11>
  <16>
][
  <21>
  <26>
  <31>
  <36>
][
  <41>
  <46>
  <51>
  <56>
]]
```

```
REQUEST: a5
ANSWER:
shape: <3 1 5>
[[
  <1 2 3 4 5>
][
  <21 22 23 24 25>
][
  <41 42 43 44 45>
]]
```

```
||=======================================
REQUEST: a7
ANSWER:
shape: <3 4 5>
[[
  <1 2 3 4 5>
  <6 7 8 9 10>
  <11 120 130 140 15>
  <16 17 18 19 20>
][
  <21 22 23 24 25>
  <26 27 28 29 30>
  <31 320 330 340 35>
  <36 37 38 39 40>
][
  <41 42 43 44 45>
  <46 47 48 49 50>
  <51 52 53 54 55>
  <56 57 58 59 60>
]]
```

References

[1] J. Backus, Can Programming be liberated from the von Neumann style: A functional style and its algebra of programs, Communications of the ACM 22, no. 8, pp. 613-641, Aug. 1978.

[2] J. Bird, P. Wadler, Introduction to Functional Programming, Prentice Hall, 1988

[3] C.A.R. Hoare, An Axiomatic Basis for Computer Programming, Communications of the ACM, vol.12, no. 10, 1969.

[4] K.E. Iverson, A Programming Language, John Wiley and Sons, 1962.

[5] L.M.R. Mullin, *A Mathematics of Arrays*, PhD thesis, Syracuse University, 1988.

[6] L.M.R. Mullin, G. Gao, M. Santavy, & B. Tiffou, Formal Program Derivation for a Shared-Memory Architecture: LU-Decomposition, McGill University, TR in progress, May 1990.

[7] J.C. Reynolds, Reasoning About Arrays, Communications of the ACM 22, no. 5, pp. 290-299, May 1979.

[8] D.B. Skillicorn, Architecture-Independent Parallel Computation, Queen's University, TR no. ISSN-0836-0227-90-268, Mar. 1990.

[9] D.A. Turner, Miranda: a non-strict functional language with polymorphic types, in J.-P.Jouannaud, editor, Functional Programming Languages and Computer Architecture, Springer-Verlag, 1985.

[10] D.A. Turner, An overview of Miranda, SIGPLAN Notices, Dècember 1986.