Introducing array-like structures in an expression-based toy-programming language

Lapo Toloni
l.toloni-at-studenti.unipi.it

March 1, 2020

1 Introduction to the new constructs

The focus of this project is to introduce in the language array-like structures and operations to play with them. All along the codebase[1] and in this report we will refer to these linear structures as **vectors**. In particular the newly implemented features are:

- Declaration and evaluation of a vector of expressions. Multidimensional vectors are supported too.
  As an example we are now allowed to write and have evaluated:

```
[1,2,3]
[let x = 5 in if ( x < 5) then x-2 else x+1, 1*4-2, 42]
let base = 1 in [base+1, base+2, base+3]
```

  The only constraint is that every element must evaluate to a value of the same type.

- Vector access. For example:

```
let ve = [1,2,3,4,5,6] in ve[0] - ve[5]
let mat = [[1,2,3],[4,5,6],[7,8,9]] in mat[1][2]
```

  Note that in this status any check is performed on indices out of bounds. LLVM library calls returns 0 when we access a not allocated piece of memory

- Vector update. For example:

```
var mat = [[1,2,3],[4,5,6],[7,8,9]] in
  mat[0]:= [42,42,42]
```

- vector concatenation. For example:

[1] https://github.com/lapotolo/languages-interpreters-compilers-unipi

```
let ve1 = [0,1,1,2] in
let ve2 = [3,5,8] in
  ve1 ++ ve2  // gives the list [0,1,1,2,3,5,8]
```

- syntactic sugar to ease the declaration of a vector. For example:

```
[[1,2,3]] times 10 // gives a list of 10 lists whose elements are
                   // [1,2,3]
```

- shortcircuit evaluation model for boolean AND and OR. For example:

```
(10 < 3) & (1 = 1)  // right-hand-side is not evaluated
(5 != 3) | (false)  // right-hand-side is not evaluated
```

- evaluation of a sequence of expressions. For example:

```
var i = 0 in
var c = 1 in
while (i < 10) do
  seq
  c := c * c + i;
  i := i + 1.
```

- support for the modulo binary operation. For example:

```
42 mod 9
```

2 Scanner

Here the extension is trivial. The following table subsumes the addition to the lex scanner:

lexeme	token	associated semantics
++	CONCAT_KW	vector concatenation
times	TIMES_KW	syntactic sugar for vector declaration
&	AND_SC	short-circuited AND
||	OR_SC	short-circuited OR
seq	SEQ_KW	expression sequence
mod	MOD	modulo operation

Moreover, square brackets are used to delimit vectors and to access them; commas separate vector elements, semicolons separate expressions in a sequence while dots terminate an expression sequence.

3 Parser

Regarding the syntactic analysis phase, the only non-trivial extensions are those implemented to parse vectors and related operations.
The LALR(1) parser built with Bison is enhanced with new productions recognizing vector of expressions. An expression may now derive a new non-terminal symbol **vect**$_e lem, delimitedbysquarebrac$

```
expr : ...
     | '[' vect_elem ']'    { $$ = make_vect($2); }
```

The computation associated with this production is the call to **make_vect()**, an internal function taking as argument a value of type **struct expr_vect** and producing a value of type **expr** with type tag **VECT**.

struct expr_vect is a new possible form of the internal struct used to represent a node of the abstract syntax tree. This kind of node holds the result of the evaluation of the current expression and a pointer to the following expression of the vector.

Four other new productions are related with it. These encode vector elements and vector continuation (and termination):

```
vect_elem: expr vect_elem_continuation
         | %empty

vect_elem_continuation: ',' expr vect_elem_continuation
                      | %empty
```

The two %empty productions produces a **NULL** and are respectively used to deal with empty vectors and termination of vectors. More interestingly both the first productions of the two rules have as associated computation a call to **make_expr_vect()** that is the function that produce the value of type **struct expr_vect** that will be eaten by the function **make_vect()**.

The possibility of binding a vector to a name using the **let** construct or of storing it in memory as a variable using the **var** construct comes for free because of a vector of expressions is an expression itself.
To support the standard operations of arrays we add two more possible productions starting from the non-terminal expr:

```
expr: ...
    | expr '[' expr ']'
    | expr '[' expr ']' ASSIGN_OP expr
```

By not specifying that we require an IDENT token standing in front of a square bracket and a VAL token to index a vector, we keep the representation homogeneus at the syntactic level. In addition, we let the back-end of the compiler decide how to interpret malformed

phrases.
For example, if the first expression in the vector access rule does not evaluate to a valid name, then the compilation will fail because of the failure of the C function that resolves the names in the environment (arguments will not match!).
A less typing intensive form used to obtain an expression yielding a vector of a desired size is associated to the production.

```
expr: ...
    | '[' vect_elem ']' TIMES_KW expr
```

The last thing to note about syntactic design choices is related to how we can sequence a block of expressions:

```
expr: ...
    | SEQ_KW expr_sequence

expr_sequence : expr expr_seq_cont

expr_seq_cont : ';' expr expr_seq_cont
              | '.'
```

We can note that this solution is almost the same as the one used for vectors of expressions. However to keep a functional flavour in the language we decided to avoid using curl brackets to enclose a sequence of expressions.

Internally at the level of the abstract syntax tree a sequence of expressions is represented just as a vector of expression. Deeper in the back-end, when LLVM API comes into play, this two features cannot be the same anymore since a vector is represented through a LLVMArrayType and so every element must evaluate to an object of the same type, while in a sequence every element can evaluate to a different type.
Finally note that every possible check on types (integers, booleans, vectors) is left to the back-end. It is a waste of time to make the front-end more complex to deal with them while being quite sure that the chosen back-end library will perform a better job.

4 Code Generation

The code generation phase consists in translating to some intermediate representation the tree structure generated by the parser. In this project this is achieved via calls to the LLVM API.
In particular to generate the IR code for a vector of expressions it is needed to:

- compute the length of the vector;
- create a C array to temporarily keep in memory the result of the evaluation of each expression in the vector together with its IR code;

- use a LLVMBuildAlloca call to allocate the required space to store that vector of expressions in memory and obtain the base address of the allocated space;
- for each expression in the vector (already evaluated and whose result is stored in the C array "expressions"):
  - use LLVMBuildInBoundsGEP to compute the offset indicating the location of memory where to store the current expression. The return value of LLVMBuildInBoundsGEP is the pointer to the memory location indicating the next free block in a contiguous segment of memory bounded by $base\ address$ and $base\ address + vector\ length$.
  - use LLVMBuildStore to generate code to actually store that expression to the correct location.

Both for vector accesses and vector updates the pattern to generate the code is similar to the one used to create a vector of expressions. In fact in LLVM, every time we have to deal with array-like structures, the crucial point is the computation of the correct pointer to the required element indicated by a given index. This is always done via LLVMBuildInBoundsGEP and similar library calls.
Code generation for the "seq" construct is based on a simple loop over the internal C structure holding the sequence of expressions. In this case we do not have to generate code to store contiguously in memory the results of each evaluation since every expression is evaluated but only the result of the evaluation of the last expression in the sequence is returned.
Then we extended the "codegen_expr" function for the "bin_op" case to make the compiler capable of generating IR code for short-circuited AND and OR and for vector concatenation.
Let's take as an example the implementation of the short-circuited AND. We have to create LLVM basic blocks that encode the fact that the first operand is true or false and one last basic block representing the continuation of the program. Since we have to evaluate the second operand only in the case that the first is true, we conditionally branch on the result of the evaluation of the left-hand-side expression.
If it is true, then we continue the computation in the basic block that contains the calls to generate the code for the right-hand side. Otherwise, from the branching point the computation follows a path going through an empty basic blocks, encoding the fact that the left-hand-side expression was false and no code has to be generated. Both blocks have a single outgoing edge sinking in the continuation basic block. The last thing to note is the presence of a LLVMBuildPhi call as the first instruction of the continuation basic block.
Vector concatenation instead follows a pattern similar to the one used for vector creation. Some more effort is required to compute the length of the concatenated vector starting from an integer that is wrapped into a LLVMValueRef. Next, we perform two loops to draw first from the left-hand-side vector and then from the right-hand-side. The body of these two loops is almost the same as the one used in the vector creation part. The single difference regards the use of LLVMBuildStructGEP and LLVMBuildLoad to load from

memory the elements of the two vectors that will be concatenated.
Excluding future optimization passes, these sets of load are present even if the two vectors were not previously stored in memory, since the model of computation is eager and every expression is always evaluated as soon as it is encountered. Note that a unique form of laziness is introduced in the language through short-circuited boolean operations.

5 An example Program

We list a simple program to show the new features of the language.

5.1 Computation of the n-th Fibonacci number

This is a standard example of a dynamic programming algorithm to compute in linear time and space the n-th Fibonacci number:

```
let n = 10 in
var i = 2 in
var x = [0] times n in
  seq
    x[0] := 0;
    x[1] := 1;
    while i < n do
      seq
        x[i] := x[i-1] + x[i-2];
        i := i+1.;
    x[n-1]. \\ n := 10 implies x[n-1] = 34
```

6 Optimization and static analysis

LLVM provides optimization and static analysis through an object called LLVMPassManager. Optimizations are realized as passes traversing some portion of the program to gather information or to transform it. Transform passes mutate the generated IR code mantaining its semantics. One common optimization pass is the one provided by **LLVMAddInstructionCombiningPass**. A first example of the transformation performed by this pass is the following:

```
var x = [1,2,3] in
  seq
    x[0] := x[1];
    x[0].

generating code...
```

```
define i32 @main() {
entry:
  %x = alloca [3 x i32]*
  %0 = alloca [3 x i32]
  %1 = getelementptr inbounds i32, [3 x i32]* %0, i32 0
  store i32 1, i32* %1
  %2 = getelementptr inbounds i32, [3 x i32]* %0, i32 1
  store i32 2, i32* %2
  %3 = getelementptr inbounds i32, [3 x i32]* %0, i32 2
  store i32 3, i32* %3
  store [3 x i32]* %0, [3 x i32]** %x
  %4 = load [3 x i32]*, [3 x i32]** %x
  %5 = load [3 x i32]*, [3 x i32]** %x
  %6 = getelementptr inbounds [3 x i32], [3 x i32]* %5, i32 0, i32 1
  %7 = load i32, i32* %6
  %8 = getelementptr inbounds [3 x i32], [3 x i32]* %4, i32 0, i32 0
  store i32 %7, i32* %8
  %9 = load [3 x i32]*, [3 x i32]** %x
  %10 = getelementptr inbounds [3 x i32], [3 x i32]* %9, i32 0, i32 0
  %11 = load i32, i32* %10
  ret i32 %11
}

generating optimised code...

define i32 @main() {
entry:
  %0 = alloca [3 x i32], align 4
  %1 = getelementptr inbounds i32, [3 x i32]* %0, i64 0
  store i32 1, i32* %1, align 4
  %2 = getelementptr inbounds i32, [3 x i32]* %0, i64 1
  store i32 2, i32* %2, align 4
  %3 = getelementptr inbounds i32, [3 x i32]* %0, i64 2
  store i32 3, i32* %3, align 4
  %4 = getelementptr inbounds [3 x i32], [3 x i32]* %0, i64 0, i64 1
  %5 = load i32, i32* %4, align 4
  %6 = getelementptr inbounds [3 x i32], [3 x i32]* %0, i64 0, i64 0
  store i32 %5, i32* %6, align 4
  ret i32 %5
}
```

```
running...
-> 2
```

Note that the two virtual registers **%4** and **%5** holds the same address in the non optimised code. In the end, by avoiding repeated computation of already known addresses and reducing registers spilling, we spare five virtual registers.
The same pass also enables the elimination of redudant variables as if it is performing a live variable data-flow analysis:

```
var n = 10 in
var c = 1 in
var b = 0 in
var a = 0 in
seq
  while a < n do
    seq
      b:= a+1;
      c:= c+b;
      a:= b*2.;
  c.

generating code...

define i32 @main() {
entry:
  %a = alloca i32
  %b = alloca i32
  %c = alloca i32
  %n = alloca i32
  store i32 10, i32* %n
  store i32 1, i32* %c
  store i32 0, i32* %b
  store i32 0, i32* %a
  br label %cond

cond:                                             ; preds = %body, %entry
  %0 = load i32, i32* %a
  %1 = load i32, i32* %n
  %2 = icmp slt i32 %0, %1
  br i1 %2, label %body, label %cont

body:                                             ; preds = %cond
```

```
  %3 = load i32, i32* %a
  %4 = add i32 %3, 1
  store i32 %4, i32* %b
  %5 = load i32, i32* %c
  %6 = load i32, i32* %b
  %7 = add i32 %5, %6
  store i32 %7, i32* %c
  %8 = load i32, i32* %b
  %9 = mul i32 %8, 2
  store i32 %9, i32* %a
  br label %cond

cont:                                             ; preds = %cond
  %10 = load i32, i32* %c
  ret i32 %10
}

generating optimised code...

define i32 @main() {
entry:
  %a = alloca i32, align 4
  %c = alloca i32, align 4
  %n = alloca i32, align 4
  store i32 10, i32* %n, align 4
  store i32 1, i32* %c, align 4
  br label %cond

cond:                                             ; preds = %body, %entry
  %storemerge = phi i32 [ %6, %body ], [ 0, %entry ]
  store i32 %storemerge, i32* %a, align 4
  %0 = load i32, i32* %n, align 4
  %1 = icmp slt i32 %storemerge, %0
  br i1 %1, label %body, label %cont

body:                                             ; preds = %cond
  %2 = load i32, i32* %a, align 4
  %3 = add i32 %2, 1
  %4 = load i32, i32* %c, align 4
  %5 = add i32 %4, %3
  store i32 %5, i32* %c, align 4
```

```
  %6 = shl i32 %3, 1
  br label %cond

cont:                                             ; preds = %cond
  %7 = load i32, i32* %c, align 4
  ret i32 %7
}

running...
-> 12
```

We can see how the two variables a and b are merged in the single variable/register a in the optimized code.
In addition the LLVMAddInstructionCombiningPass forces 4-byte aligned address, the replacement of multiplications and divisions with shifts and other simple peephole optimizations.
Another possible pass to add is LLVMAddCFGSimplificationPass. This extension makes LLVM delete unreachable blocks by simplifying the control flow graph of the program, as we can see in:

```
let x = 5 in
if x < 2 then x+1 else x-1

generating code...

define i32 @main() {
entry:
  br i1 false, label %then, label %else

then:                                             ; preds = %entry
  br label %cont

else:                                             ; preds = %entry
  br label %cont

cont:                                             ; preds = %else, %then
  %0 = phi i32 [ 6, %then ], [ 4, %else ]
  ret i32 %0
}

generating optimised code...

define i32 @main() {
```

```
entry:
  ret i32 4
}
```

Since the value of x is known at compile time, it can be propagated forward through the program. This implies that the compiler is be able to infer that the ”then” branch will never be an actual computation path and so that portion of code is useless and will not be generated.