Supporting Vector Expressions and Patterns in MLton

Student : Krishna Ravikumar (kr5748@rit.edu)

Advisor : Dr. Matthew Fluet (mtf@cs.rit.edu)

Department of Computer Science, Rochester Institute of Technology

May 2016

Abstract

Standard ML (SML) is a strict, statically-typed functional programming language. Two of the most popular compilers for SML are MLton [1] and Standard ML of New Jersey (SML/NJ) [2]. In the SML language, vectors are homogeneous, immutable arrays. Vectors are a standard feature of SML'97, but SML/NJ also has special syntax for vector expressions and vector patterns. In MLton, as with SML'97, vectors can be created only through the Vector structure [5], and cannot be pattern-matched. The project aims to implement support for vector expressions and patterns similar to what SML/NJ offers, and introduce representation for such vectors internally so that optimizations on them are possible.

1 Introduction

Modern compilers, in addition to supporting the complete specification of a language, sometimes offer additional features exclusive to their compilers based on various factors including the new feature's usefulness, implementation overhead, community feedback etc. In the context of the SML functional programming language, an example of such

special features are vector expressions and vector patterns. The SML/NJ compiler supports vector expressions and patterns that enable creation of vectors (homogeneous, immutable arrays), and allow pattern-matching on them (vector patterns).

The goal of this project is to extend the MLton implementation of SML to support vector expressions, and subsequently, vector patterns. The introduction of vector expressions and patterns should not result in any backwards compatibility issues since they are a standalone feature independent of any existing syntax or semantics. This report provides an outline of the significance of the project, including background information pertaining to the working of compilers in general as well as the MLton compiler, and an overview of the efforts carried out towards achieving the goal. There is also an account of the various pain points that might be of concern when undertaking such a project, as well as some details of future work that could be carried out to further satisfy the goal of the project.

2 Background

2.1 Standard ML (SML)

Standard ML is a strict, statically-typed, functional programming language with type inference, abstract data types, a sophisticated module system, garbage collection, and many other features. It is popular among compiler writers and is also quite widely used in the programming language theory world. The language is defined by a formal specification given as a set of typing rules and operational semantics [3]. The original specification was revised in 1997 [4], and this revised specification (SML'97) is what current SML compilers adhere to.

2.2 MLton

MLton is an open-source, whole-program, optimizing compiler for the Standard ML programming language. MLton adopts the SML'97 specification including an implementation of the Basis Library specification. The whole-program approach that MLton applies to the compilation process allows it to provide advanced programming language features, while still having superior performance. This can be attributed to the fact that whole-program compilation enables MLton to have enough information that permits it to eliminate or mitigate the performace costs of using advanced programming language features. MLton also supports the compilation of large programs (including itself), and handle large inputs, while maintaining a reasonable compilation time.

2.3 MLton architecture

The compilation framework of MLton is best described as a series of passes that transform an input program from one representation to another while performing some task of significance. The entire compilation process is a sequence of five major steps, namely scanning, parsing, type-checking (static analysis), optimization and finally code generation.

Translation Passes	*Intermediate Language*	*Optimization Passes*
	Source	
FrontEnd		
	AST	
Elaborate		
	CoreML	CoreMLSimplify
Defunctorize		
	XML	XMLSimplify
Monomorphise		
	SXML	SXMLSimplify
ClosureConvert		
	SSA	SSASimplify
ToSSA2		
	SSA2	SSA2Simplify
ToRSSA		
	RSSA	RSSASimplify
ToMachine		
	Machine	
Codegen		

Figure 1: Overview of MLton's architecture

The overall structure of the MLton compiler is represented as a table in Figure 1. It shows the various intermediate languages that an input program is turned into, as well as the transformation passes that turn one intermediate representation into another and optimization passes that operate on a certain representation and carry out optimization procedures. The optimization passes themselves are a series of steps that aim to address a specific optimization criteria. The MLton website [1] is a good place of reference to further understand the importance of each of these passes and the functions that they perform.

2.4 MLton and Vectors

MLton follows the SML'97 specification closely with respect to the implementation of vectors. Hence, currently the only way to create vectors is by using the Vector structure [5] in the Basis Library. Vector expressions provide a convenient way to create vectors of relatively small sizes on the fly, by way of new syntax. The SML/NJ syntax for creation of vector expressions is represented below:

$$\#[exp_0, exp_1, \ldots, exp_{n-1}]$$

Figure 2: Vector Expression Syntax

Internally, MLton manages vectors by elaborating the creation of any vector into a set of simpler steps. To create a vector of size *n* with elements a_0 through a_n, MLton

1. creates an uninitialized array of size *n*
2. performs an array update operation *n* times with the corresponding values
3. unsafely casts the array into a vector

While this seems like a reasonable approach to handling vectors internally, the lack of a primitive Vector creation operation means that MLton loses out on optimization opportunities that might have been possible by keeping vectors in a high-level representation. Also, since vectors are immutable by nature, treating their contents as constants could provide better benefits. For example, an optimization such as common subexpression elimination could replace all occurrences of the same vector expression with a single variable holding it. Loop invariant code motion is another optimization technique that can exploit vector expressions. It can decide to move vector expressions outside of the body of a loop while preserving the semantics of the program. An important observation is that regardless of the presence of a vector primitive, the above elaboration is necessary as we move on to the later stages of the compiler.

3 Implementation

3.1 Design Plan

Before performing changes to a vast code-base, it is essential to plan ahead. Assessment of the code-base to determine which areas would require modifications and understanding their working were tasks of importance. It is also imperative that any additions and modifications to existing code does not cause any backwards compatibility issues. Since the idea is to introduce a new syntax (unrelated to existing ones) and only handle the use of this syntax in a special manner, there is not much threat in terms of backwards compatibility issues. The implementation was carried out in stages, the details of which is explained in the following sections.

3.2 Front-end changes

For addition of the new vector expression syntax, changes would have to be made to the scanner portion of the compiler to identify the Hash-bracket token (#[ ) for vector expressions. This token is currently not in use and is unique to vector expressions. The parser would also require adjustment to identify the occurrence of this new token and parse the arguments appropriately. MLton makes use of some tools that generate the code for its scanner and parser. Specifically, the lexer is generated by the lexical analyzer generator MLLex. MLLex is modeled after the Lex lexical analyzer generator and takes as input the lex language as defined in the ML-Lex manual, and outputs a lexical analyzer in SML. The parser is generated by MLYacc, a parser generator for SML modeled after the Yacc parser generator. The initial step would thus be to modify the specifications of these tools.

Firstly, the lexer specification was modified to include a new token Hash-bracket (#[ ). This token would be used to parse vector expressions which could be created as a list of

expressions (of the same type) enclosed between a hash-bracket and a right-bracket. Next, the parser's grammar was modified to introduce a transition from the initial state to a new state on occurrence of the Hash-bracket token. In the new state, the parser parses a list of zero or more, comma-delimited expressions. On encountering a right-bracket, the parser concludes the parsing of the vector expression and goes back to the initial state. This concludes the changes required to support the vector expression syntax.

3.3 AST and CoreML

The front end pass translates the source program into an abstract syntax tree (AST) intermediate representation. Following this, the Elaborate pass works on the AST representation, performing type inference and type checking and translating into the CoreML intermediate representation. When the parser recognizes a vector expression, it needs to create a corresponding AST node that represents the vector. Similarly, the Elaborate pass has to turn any encountered vector expressions from the AST into a corresponding expression in the CoreML representation. Hence, modifications were made to both the intermediate representations to accommodate new vector expression representations and support type inference and type checking of vector expressions. Both passes propagate the vector expression as the application of the appropriate primitive to the arguments. The similarity in syntax between vector expressions and lists proved to be helpful in identification of the required changes to the above passes.

3.4 Defunctorize

The defunctorize translation pass works on the CoreML intermediate language and turns it into the XML intermediate language. It was previously noted that the processing of vectors internally appeared to be quite similar to how lists were handled. List expressions were eliminated in the defunctorize pass of the compiler into simpler

expressions (the explicit application of the *cons* and *nil* constructors). Hence it was decided that elimination of the vector expressions in this pass would be a good starting point. The later parts of the compiler would not need to know about vector expressions yet, and this allowed for validation of all the changes performed so far, by way of compiling a toy example program that used vector expressions.

The defunctorize pass was altered to generate the appropriate instructions in the XML intermediate representation on encountering the vector expressions. This process was described in previous sections as the way MLton handled vectors internally. The generated code is the application of the appropriate XML primitive to the correct arguments to perform the following steps:

1. create an uninitialized array of the size of the arguments
2. perform an array update of all the positions with their corresponding arguments
3. unsafely cast the array into a vector

A small test program was written that created a vector using the new syntax for vector expressions, and printed out an element of the vector using a Vector.sub operation. The changes were validated by the successful compilation of the test program and the exhibition of the correct behavior by the generated executable. The generated intermediate files were also checked to ensure that the correct instructions were generated by the defunctorize pass for vector expressions.

3.5 The Vector_vector Primitive

With vector expressions being successfully recognized and translated in the defunctorization stage, the next step was to introduce a primitive to represent the creation of vector expressions internally. Introducing a new primitive needs to be

handled carefully to ensure that every area of the code base is modified to accommodate it. The main concern is that the various passes use pattern-matching to identify a primitive and take appropriate action, and most of them contain a wild-card (_) case to perform a conservative action when encountering unrecognized primitives. Forgetting to alter these pattern-match cases might mean that some pass could potentially perform the wrong action when encountering the newly introduced vector primitive.

To alleviate the difficulty of the above task, the renaming of an existing primitive was performed first, to identify every position of the code-base to be modified. The internal primitive used to create an uninitialized array was renamed to more appropriately represent the function it performs (Array_array → Array_uninit). Simultaneously, a new primitive (Vector_vector) was also introduced and propagated across the various phases of the compiler. This new primitive took an arbitrary number of arguments, and its function was to create a vector containing the arguments as its elements. The newly created Vector_vector primitive is 'functional' in nature, meaning that it can be freely promoted to a global variable under favorable circumstances.

3.6 Delaying elimination of Vector Expressions

The elimination of vector expressions in the defunctorize translation pass offered a good check-point for testing prior changes. However, optimally such a task would have to be delayed until after some optimization passes, so that they can exploit the high-level representation of vector expressions in their optimization procedures. The ideal position to perform this elimination would be in the SSASimplify optimization pass, that works on the Static Single Assignment (SSA) intermediate representation.

Since the elimination of vector expressions is not set to happen until the SSA stages, the

defunctorize stage was once again modified to mimic the behavior of its predecessors. The pass now propagates vector expressions as an application of the Vector_vector primitive to its arguments. The subsequent Monomorphise pass did not require any modification as its existing behavior was to only propagate uninteresting primitives down the chain.

3.7 SSA Simplify – The implementVectors pass

The SSASimplify optimization pass is a set of optimizations working on the SSA intermediate representation. Some of the optimizations tasks include constant propagation, redundancy elimination, inlining, unused code removal etc.

CombineConversions	LocalFlatten
CommonArg	LocalRef
CommonBlock	LoopInvariant
CommonSubexp	Redundant
ConstantPropagation	RedundantTests
Contify	RemoveUnused
Flatten	SimplifyTypes
Inline	Useless
IntroduceLoops	PolyEqual
KnownCase	PolyHash

Figure 3: SSASimplify Optimization Passes

Figure 3 shows the various optimizations carried out as part of this pass. In order to eliminate the vector expression primitive, an additional pass is added to the above chain as an SSA → SSA optimization. This pass, called 'ImplementVectors', performs the task of eliminating vector expressions in a manner similar to what defunctorize used to carry out before.

The implementVectors pass walks through the list of globals, transforming each vector expression occurrence as and when encountered. In transforming the code in the various function blocks, the pass adopts a more efficient approach. It first performs a read only run-through of all the functions, scanning them for any occurrence of the Vector_vector primitive. If any function or block is found to have the primitive, they are marked with a corresponding label. After this process, the pass scans all functions again, transforming them as is, and only looking at the contents of those blocks with the label in order to eliminate the vector expressions. This approach is more efficient since we do not have to look at the contents of each of the functions or blocks, except during the initial pass, which turns out to be fast anyway since it is a read-only pass.

3.8 Closure Convert

It was initially deemed that closure convert pass did not need any modifications since it propagated unknown primitives and their applications down the chain without any modifications. However, the closure conversion process was necessary to ensure that any vector expressions that contained functions inside them were properly handled.

A second toy example was written that created a small vector of functions. A Vector.sub operation was performed to fetch one of these functions and to print out the result of calling it with a preset argument. This example failed with the current setup of MLton, and acted as a good validation criteria for ensuring that the right changes were made to closure convert.

Closure convert was initially modified to accommodate vector expressions in a manner similar to how it handled arrays. The change involved modifying the abstract-value

signature. This seemed like a valid approach, but the compiler still failed to compile. This was because of an oversight of the closure convert pass itself, which also needed adjustment to support vector expressions. Once this was done, the globalize method of this pass was tweaked to ensure that vector expressions were being promoted to be a global variable under the right conditions.

3.9 Positioning of optimization passes

The implementVectors pass was initially positioned as the first optimization in the SSA Simplify chain. This is not very beneficial since in effect, none of the optimizations see vector expressions anyway. But it was again a good starting point to ensure the correct functionality of the pass before moving ahead. Once the functionality was validated, the positioning of the implementVectors pass had to be changed. The ideal position would be at the end of the chain once all the other optimization passes have been completed. This seemed like a straightforward change since none of the other passes depended on vector expressions being transformed - they were oblivious to such expressions.

4 Implementation Difficulties

4.1 Errors and Bugs

The initial idea was to move the implementVectors all the way to the end of the chain and see the resultant behavior. It was expected that the compiler should work as intended, or throw up some internal compiler errors. However, a more difficult to diagnose problem came up. The compiler was able to compile itself, as well as the toy example and generated an executable. The executable however did not have the intended behavior. Such bugs are more dangerous since they are hard to detect and harder to fix.

In order to find out the root cause of this error, a binary search like approach was adopted with respect to the position of the implementVectors pass to find out which position caused this issue. Initially, it was determined that moving the implementVectors pass past the constantPropagation optimization pass resulted in the above behavior. It did not seem like constant propagation was doing anything out of the ordinary with vector expressions, and hence this seemed like a dead end. Constant propagation was modified anyway to propagate the length of such vector expressions as constants. Though it was an unrelated change, it was convenient to make changes while closely observing the pass.

A peculiar behavior was that the seemingly unrelated change to constant propagation now resulted in no errors on moving the implementVectors pass. This was a false assumption, since the toy example did work properly, but later once the compiler was modified to make use of vector expressions internally, it failed to compile and build itself. This brought things back to square one.

4.2 Fixing the Positioning Bug

To resolve the above mentioned bug, the intermediate files generated after each pass were preserved. This can be done by invoking compilation of a program with special flags. The files generated for the small example program were scanned to ensure that each pass was behaving as intended and did not create any issues. All the intermediate files for the SSA simplify pass seemed to be correct, but the issue was still not resolved. To probe this further, the passes that followed the SSA simplify pass were looked into in greater depth.

The underlying cause was identified when looking at the later stages of the compiler

(RSSA to machine translation). The closure convert pass treated vector expressions to be 'functional' in nature and allowed them to be promoted to a global variable. This also meant that the corresponding length of the vector was also being promoted to a global. This promotion is also valid since vectors are immutable and are of fixed lengths. The incorrect behavior happened in the RSSA to machine code translation pass, wherein the instruction storing the vector length, now propagated to a global, was being dropped.

Dropping such an instruction was necessary in prior architectures of MLton that did not have an SSA representation, but had a continuation-passing style (CPS) representation instead. The CPS representation was being directly translated into machine code, and hence certain instructions such as binding integer constants to variables had to be dropped. This was appropriate because we would want machine code to have integer constants as an immediate operand rather than having to look them up from some position in memory each time. Since SSA was introduced, this behavior should have been removed, and this seemed to cause the issue since trying to access a vector's element would result in a check to see if the index is in bounds first, and this check would now fail since the vector length was being dropped. Once this behavior was changed, the implementVectors pass was moved to the end of the optimization chain as desired.

4.3 Internally using Vector Expressions

MLton can now make use of vector expressions internally to create vectors of small sizes. Vectors of sizes zero through six were previously being created internally using the basis library. The signature of the vector structure being used was modified to ensure that these were no longer basis library calls, but were vectors being created using vector expressions. This change results in a noticeable drop in the size of the generated binary,

since small vectors were widely used in the compiler.

```
fun new3 (x0, x1, x2) =
   tabulate (3,
             fn 0 => x0
              | 1 => x1
              | 2 => x2
              | _ => Error.bug "Vector.new3")
```

Figure 4: Vector.new3 implementation - Before

```
val new3 = fn (x0, x1, x2) => #[x0, x1, x2]
```

Figure 5: Vector.new3 implementation - After

The implementation of Vector.new3, the operation for creating a new vector with 3 elements is shown in figures 4 and 5. It is observed that the vector creation process is much simpler when vector expressions are used. This can be attributed to the fact that the latter generates straight line code, whereas the former generates a loop, which is inherently harder for the compiler to handle. Also, the additional overhead of having to specify an 'impossible' case in the former implementation is avoided when using vector expressions.

5 Currently Accomplished Goals

5.1 Results

The goal of the project was to support vector expressions and vector patterns in MLton. The current build of MLton supports vector expressions completely. It provides users with the new syntax for vector expressions, as well as makes use of them internally. This results in a smaller binary for the compiler. The new MLton executable is around 0.8 megabytes smaller than the fork initially used to start this project (32803 → 32005 kilobytes).

```
val _ =
    let
        fun loop n =
          let
              val v = #[0,1,2,3,4,5,6,7,8,9]
              val _ = print(Int.toString(Vector.sub (v, 5)))
          in
              if n < 500000
              then loop (n + 1)
              else ()
          end
    in
        loop 0
    end
```

Figure 6: Test example – use of vector expressions

```
open Vector
val _ =
    let
        fun loop n =
          let
              val v = tabulate(10, fn i => i)
              val _ = print(Int.toString(Vector.sub (v, 5)))
          in
              if n < 500000
              then loop (n + 1)
              else ()
          end
    in
        loop 0
    end
```

Figure 7: Test example – use of vector tabulate

Figures 6 and 7 show example programs that create a vector using vector expressions and using vector tabulate respectively, and print out an element 500,000 times. Both the programs were run and the following data was gathered from the example. All values are averaged over 20 runs. GC refers to the Garbage Collector.

Property	Vector expression	Vector tabulate
Total running time (ms)	2,424	2,467
GC- # of copies	734	917
GC – bytes copied	9,976,528	12,405,176
Total bytes allocated	128,014,368	160,014,368

Figure 8: Runtime and Memory comparisons

From Figure 8, we observe that the garbage collector works overtime in the case where vector tabulate was used. The total bytes allocated has also fell when vector expressions were used. This is because creation of the same vector (with the same elements) using vector expressions reuses the previously created vector in memory, resulting in lesser load on the garbage collector as well as reducing the total memory used.

5.2 Synergy with other ongoing projects

Having vector expressions would work well with another ongoing project in MLton – loop optimizations. Loop unrolling (both partial and complete) could result in vector expressions being exposed, which are then elevated to being global constants. If loop unrolling is performed early enough, vectors created using Vector.tabulate of a constant number will be turned into straight line code. This allows for another optimization to be performed that looks for the pattern of expressions used to create a vector (creating an uninitialized array → updating its values → casting it to a vector), and substitutes them for a vector expression (an “unimplementVectors” pass).

6 Future Work

To extend the work done in this project, the first steps would be to find additional optimization opportunities on vector expressions. The knowledge of vectors being fixed in length and immutable (meaning their contents are constant) could be exploited to more aggressively optimize operations on vectors.

Vector expressions also form a strong foundation upon which Vector patterns can be incorporated. Vector patterns refer to the use of vector expressions in pattern-matching. Supporting vector patterns would require work on the match compile portion of the compiler, that has the logic for pattern matching. Specification of what constitutes a pattern match, and how such matching can be performed efficiently without too much computational overhead are areas to work on.

7 Conclusions

This report gives an overview of the MLton compiler and its architecture and the planning and efforts involved in modifying its code base. The new feature of vector expressions was implemented successfully, and is now internally being used by the compiler. The implementation difficulties outlined and the solutions for the problems faced are good lessons in software development in general. Support for vector patterns was intended to be an extended milestone, which remains unfinished due to the above sighted implementation hardships. The performance comparisons in Figure 8 also suggest that the new feature pays for itself by means of decreasing the size of the compiler executable. The addition of this feature is also a step towards being consistent with other modern SML compilers (SML/NJ).

Bibliography

[1] MLton (http://mlton.org).

[2] Standard ML of New Jersey (http://www.smlnj.org).

[3] Milner R., Tofte M., Harper R. “Definition of Standard ML”, *The MIT Press*, 1990.

[4] Milner R., Tofte M., Harper R., MacQueen D. “The Definition of Standard ML” Revised, *The MIT Press*, 1997.

[5] The Vector Structure (http://sml-family.org/Basis/vector.html), *The Standard ML Basis Library*.