

# *Master Thesis*

## An LLVM Backend for Accelerate



Christian-Albrechts-Universität zu Kiel  
Department of Computer Science  
Programming Languages and Compiler Construction

student: **Timo von Holtz**  
advised by: Priv.-Doz. Dr. Frank Huch  
Assoc. Prof. Dr. Manuel M. T. Chakravarty

Kiel, July 15, 2014



# Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

July 15, 2014

---

Timo von Holtz



# Abstract

Implementing parallel computations is traditionally done in C or FORTRAN. This allows great control over the details of execution. Unfortunately, this also comes with the complexities of using a low-level language. To make this easier, Chakravarty et al. have developed Accelerate, an embedded array language for computations for high-performance computing in Haskell. The main execution target of this EDSL is GPUs via CUDA.

McDonell started working on an alternative implementation using LLVM. I contribute to this work by implementing a different approach to code-generation for LLVM using quasi-quotation. I use this quasi-quoter to implement the skeletons in the LLVM backend.



# Todo list

write about execution, but maybe wait for trevor to write stuff . . . . .	12
Figure: flowchart of quasiquoter implementation . . . . .	19





# Contents

<b>1. Introduction</b>	<b>1</b>
<b>2. Technologies</b>	<b>3</b>
2.1. LLVM . . . . .	3
2.1.1. LLVM IR . . . . .	3
2.1.2. Vectorization . . . . .	5
2.1.3. llvm-general . . . . .	8
2.2. Accelerate . . . . .	9
2.2.1. Usage . . . . .	9
2.2.2. Representation . . . . .	11
2.2.3. Skeletons . . . . .	11
2.2.4. Fusion . . . . .	11
2.2.5. LLVM Backend . . . . .	12
<b>3. Contributions</b>	<b>13</b>
3.1. accelerate . . . . .	13
3.2. llvm-general . . . . .	13
3.3. llvm-general-quote . . . . .	13
3.3.1. Control Structures . . . . .	15
3.3.2. SSA . . . . .	16
3.3.3. syntax extensions . . . . .	18
<b>4. Implementation</b>	<b>19</b>
4.1. Quasiquote . . . . .	19
4.2. SSA . . . . .	19
4.3. Control Structures . . . . .	22
4.3.1. first version (direct) . . . . .	22
4.3.2. second version (mutable variables) . . . . .	22
<b>5. Skeletons</b>	<b>23</b>
5.1. map . . . . .	23
5.2. fold . . . . .	23
5.3. scan . . . . .	23
5.4. stencil . . . . .	24
<b>6. Conclusion</b>	<b>25</b>
6.1. Related Work . . . . .	25

<b>A. Listings</b>	<b>27</b>
<b>References</b>	<b>29</b>

# List of Figures

2.1. sum as a C function . . . . .	3
2.2. sum as a LLVM . . . . .	4
2.3. Vectorized Sum . . . . .	6
2.4. <code>foo</code> without SLP vectorization . . . . .	7
2.5. <code>foo</code> with SLP vectorization . . . . .	8
2.6. map Skeleton in C . . . . .	12
3.1. Monadic generation of for loop . . . . .	14
3.2. For Loop using <i>llvm-general-quote</i> . . . . .	16
3.3. Expanded For Loop . . . . .	17
3.4. For Loop using <i>llvm-general-quote</i> and SSA . . . . .	17
3.5. Expanded For Loop (SSA) . . . . .	18
4.1. Implementation of local value numbering . . . . .	20
4.2. Implementation of global value numbering . . . . .	21
4.3. <code>toSSA</code> . . . . .	22



## List of Tables



# 1. Introduction





## 2. Technologies

### 2.1. LLVM

LLVM[Lat02] is a compiler infrastructure written in C++. In contrast to GCC it is designed to be used as a library by compilers. Originally implemented for C and C++, the language-agnostic design (and the success) of LLVM has since spawned a wide variety of front ends: languages with compilers that use LLVM include ActionScript, Ada, D, Fortran, OpenGL Shading Language, Haskell, Java bytecode, Julia, Objective-C, Python, Ruby, Rust, Scala and C.

#### 2.1.1. LLVM IR

LLVM defines it's own language to represent programs. It uses Static Single Assignment (SSA) form.[AWZ88; RWZ88] A program is said to be in SSA form if each of its variables is defined exactly once, and each use of a variable is dominated by that variable's definition. SSA form greatly simplifies many dataflow optimizations because only a single definition can reach a particular use of a value, and finding that definition is trivial.

To get idea of how this looks like in practice, let's look at an example. Figure 2.1 shows a simple C function to sum up the elements of an array. The corresponding LLVM code is shown in figure 2.2.

The first obvious difference is the lack of sophisticated control structures. In LLVM every function is divided into basic blocks. A basic block is a continuous stream of instructions with a terminator at the end. Instructions are add, mult, call, ..., but also

```
1  int sum(int* a, int length) {  
2      int x = 0;  
3      for (int i=0;i<length;i++) {  
4          x += a[i];  
5      }  
6      return x;  
7  }
```

Figure 2.1.: sum as a C function

```

1  define i32 @sum(i32* nocapture readonly %a, i32 %length) {
2    entry:
3      %cmp4 = icmp sgt i32 %length, 0
4      br i1 %cmp4, label %for.body, label %for.end
5
6    for.body:                                ; preds = %entry, %
        for.body
7      %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ 0, %entry ]
8      %x.05 = phi i32 [ %add, %for.body ], [ 0, %entry ]
9      %arrayidx = getelementptr inbounds i32* %a, i64 %indvars.iv
10     %0 = load i32* %arrayidx, align 4
11     %add = add nsw i32 %0, %x.05
12     %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
13     %lftr.wideiv = trunc i64 %indvars.iv.next to i32
14     %exitcond = icmp eq i32 %lftr.wideiv, %length
15     br i1 %exitcond, label %for.end, label %for.body
16
17    for.end:                                ; preds = %for.body, %
        entry
18     %x.0.lcssa = phi i32 [ 0, %entry ], [ %add, %for.body ]
19     ret i32 %x.0.lcssa
20 }

```

Figure 2.2.: sum as a LLVM

call. Terminators can either be used to jump to another block (branch) or return to the calling function.

To allow for dynamic control flow, there are  $\Phi$ -nodes. These specify the the value of a new variable depending on what the last block was. The  $\Phi$ -nodes in the SSA are represented with **phi**-instructions. In LLVM, these have to precede every other instruction in a given basic block.

## Types

The LLVM type system is one of the most important features of the intermediate representation. Being typed enables a number of optimizations to be performed directly, without having to do extra analyses on the side before the transformation.

Important Types are:

- **void**, which represents no value
- integers with specified length N: **iN**
- floating point numbers: **half**, **float**, **double**, ...
- pointers: **<type> \***
- function types: **<returntype> (<parameter list>)**
- vector types: **< <# elements> x <elementtype> >**

- array types: [<# elements> x <elementtype>]
- structure types: { <type list> }

### 2.1.2. Vectorization

Modern CPUs all have SIMD units to execute an instruction on multiple datasets in parallel. Using these units is easy with LLVM. All operations (**add**, **fadd**, **sub**, ...) can be used with vector arguments the same way as with scalar arguments.

To manually exploit this can be tricky however. LLVM has multiple strategies to fuse similar instructions or tight inner loops into vectorized code.[LLV14]

#### Loop Vectorizer

The Loop Vectorizer tries to vectorize tight inner loops. To get an idea of how these work, let's look at the example from figure 2.1. To build the sum, every element of the array is added to an accumulator. Since addition is associative and commutative, the additions can be reordered. Given a vector width of 2, the sum of  $2 * n$  and  $2 * n + 1$  are calculated in parallel and then added together. In addition to this, the vectorizer will also unroll the inner loop to make fewer jumps necessary. Figure 2.3 shows the corresponding LLVM code without loop unroll, as this increases code size dramatically.

Apart from reductions, LLVM can also vectorize the following:

- Loops with unknown trip count
- Runtime Checks of Pointers
- Reductions
- Inductions
- If Conversion
- Pointer Induction Variables
- Reverse Iterators
- Scatter / Gather
- Vectorization of Mixed Types
- Global Structures Alias Analysis
- Vectorization of function calls
- Partial unrolling during vectorization

A detailed description can be found at <http://llvm.org/docs/Vectorizers.html>

```

1  define i32 @sum(i32* nocapture readonly %a, i32 %length) #0 {
2  entry:
3      %cmp4 = icmp sgt i32 %length, 0
4      br i1 %cmp4, label %for.body.preheader, label %for.end
5
6  for.body.preheader:                                ; preds = %entry
7      %0 = add i32 %length, -1
8      %1 = zext i32 %0 to i64
9      %2 = add i64 %1, 1
10     %end.idx = add i64 %1, 1
11     %n.vec = and i64 %2, 8589934590
12     %cmp.zero = icmp eq i64 %n.vec, 0
13     br i1 %cmp.zero, label %middle.block, label %vector.body
14
15 vector.body:                                        ; preds = %for.body.preheader, %
16     vector.body
17     %index = phi i64 [ %index.next, %vector.body ], [ 0, %for.body.preheader ]
18     %vec.phi = phi <2 x i32> [ %5, %vector.body ], [ zeroinitializer, %for.body.preheader
19     ]
20     %3 = getelementptr inbounds i32* %a, i64 %index
21     %4 = bitcast i32* %3 to <2 x i32>*
22     %wide.load = load <2 x i32>* %4, align 4
23     %5 = add nsw <2 x i32> %wide.load, %vec.phi
24     %index.next = add i64 %index, 2
25     %6 = icmp eq i64 %index.next, %n.vec
26     br i1 %6, label %middle.block, label %vector.body
27
28 middle.block:                                       ; preds = %vector.body, %
29     for.body.preheader
30     %resume.val = phi i64 [ 0, %for.body.preheader ], [ %n.vec, %vector.body ]
31     %rdx.vec.exit.phi = phi <2 x i32> [ zeroinitializer, %for.body.preheader ], [ %5, %
32     vector.body ]
33     %rdx.shuf = shufflevector <2 x i32> %rdx.vec.exit.phi, <2 x i32> undef, <2 x i32> <
34     i32 1, i32 undef>
35     %bin.rdx = add <2 x i32> %rdx.vec.exit.phi, %rdx.shuf
36     %7 = extractelement <2 x i32> %bin.rdx, i32 0
37     %cmp.n = icmp eq i64 %end.idx, %resume.val
38     br i1 %cmp.n, label %for.end, label %for.body
39
40 for.body:                                          ; preds = %middle.block, %for.body
41     %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ %resume.val, %middle.block ]
42     %x.05 = phi i32 [ %add, %for.body ], [ %7, %middle.block ]
43     %arrayidx = getelementptr inbounds i32* %a, i64 %indvars.iv
44     %8 = load i32* %arrayidx, align 4
45     %add = add nsw i32 %8, %x.05
46     %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
47     %lftr.wideiv = trunc i64 %indvars.iv.next to i32
48     %exitcond = icmp eq i32 %lftr.wideiv, %length
49     br i1 %exitcond, label %for.end, label %for.body
50
51 for.end:                                          ; preds = %for.body, %middle.block, %
52     entry
53     %x.0.lcssa = phi i32 [ 0, %entry ], [ %add, %for.body ], [ %7, %middle.block ]
54     ret i32 %x.0.lcssa
55 }
```

Figure 2.3.: Vectorized Sum

```

1  define void @foo(double %a1, double %a2, double %b1, double %b2, double*
    %A) {
2  entry:
3      %add = fadd double %a1, %b1
4      %mul = fmul double %add, %a1
5      %div = fdiv double %mul, %b1
6      %mul1 = fmul double %b1, 5.000000e+01
7      %div2 = fdiv double %mul1, %a1
8      %add3 = fadd double %div, %div2
9      store double %add3, double* %A, align 8
10     %add4 = fadd double %a2, %b2
11     %mul5 = fmul double %add4, %a2
12     %div6 = fdiv double %mul5, %b2
13     %mul7 = fmul double %b2, 5.000000e+01
14     %div8 = fdiv double %mul7, %a2
15     %add9 = fadd double %div6, %div8
16     %arrayidx10 = getelementptr inbounds double* %A, i64 1
17     store double %add9, double* %arrayidx10, align 8
18     ret void
19 }

```

Figure 2.4.: `foo` without SLP vectorization

## SLP Vectorizer

The goal of SLP vectorization (a.k.a. superword-level parallelism) is to combine similar independent instructions into vector instructions. Memory accesses, arithmetic operations, comparison operations, PHI-nodes, can all be vectorized using this technique. For example, the following function performs very similar operations on its inputs (`a1`, `b1`) and (`a2`, `b2`).

```

1  void foo(double a1, double a2, double b1, double b2, double *A) {
2      A[0] = a1*(a1 + b1)/b1 + 50*b1/a1;
3      A[1] = a2*(a2 + b2)/b2 + 50*b2/a2;
4  }

```

The SLP vectorizer may combine these into vector operations. Figures 2.4 and 2.5 show the the corresponding LLVM before and after SLP vectorization.

## Fast-Math Flags

In order for the Loop vectorizer to work, the operations involved need to be associative. When dealing with floating point numbers, the basic operations like `fadd` and `fmul` don't satisfy this condition. To vectorize the code regardless, LLVM has the notion of fast-math-flags. These tell the optimizer to assume certain properties that aren't true in general.

Available flags are[LLV]:

```
1  define void @foo(double %a1, double %a2, double %b1, double %b2, double*  
    %A) {  
2  entry:  
3    %0 = insertelement <2 x double> undef, double %a1, i32 0  
4    %1 = insertelement <2 x double> %0, double %a2, i32 1  
5    %2 = insertelement <2 x double> undef, double %b1, i32 0  
6    %3 = insertelement <2 x double> %2, double %b2, i32 1  
7    %4 = fadd <2 x double> %1, %3  
8    %5 = fmul <2 x double> %4, %1  
9    %6 = fdiv <2 x double> %5, %3  
10   %7 = fmul <2 x double> %3, <double 5.000000e+01, double 5.000000e+01>  
11   %8 = fdiv <2 x double> %7, %1  
12   %9 = fadd <2 x double> %6, %8  
13   %10 = bitcast double* %A to <2 x double>*  
14   store <2 x double> %9, <2 x double>* %10, align 8  
15   ret void  
16 }
```

Figure 2.5.: foo with SLP vectorization

- nnan** No NaNs - Allow optimizations to assume the arguments and result are not NaN. Such optimizations are required to retain defined behavior over NaNs, but the value of the result is undefined.
- ninf** No Infs - Allow optimizations to assume the arguments and result are not +/-Inf. Such optimizations are required to retain defined behavior over +/-Inf, but the value of the result is undefined.
- nsz** No Signed Zeros - Allow optimizations to treat the sign of a zero argument or result as insignificant.
- arcp** Allow Reciprocal - Allow optimizations to use the reciprocal of an argument rather than perform division.
- fast** Fast - Allow algebraically equivalent transformations that may dramatically change results in floating point (e.g. reassociate). This flag implies all the others.

### 2.1.3. llvm-general

There are multiple bindings to LLVM in Haskell. The most complete is `llvm-general`. [Sca13] Instead of exposing the LLVM API directly, it uses an ADT to represent LLVM IR. This can then be translated into the corresponding C++ object. It also supports the other way, transforming the object back into the ADT.

Using an ADT instead of writing directly to the C++ object has many advantages. This way, the code can be produced and manipulated outside the IO context. It is also much easier to reason about within Haskell.

llvm-general also supports optimization and jit-compilation. The optimization is controlled by a `PassSetSpec`. This offers the user 2 different options: `CuratedPassSetSpec` and `PassSetSpec`. Using `CuratedPassSetSpec` is the easier option of the 2. It offers a similar level of control as specifying `-On` at the command line. Using `PassSetSpec` gives much more control over the exact passes run, but you have to specify them one by one. Both of these specs also come with fields to specify information about the target.

## 2.2. Accelerate

Accelerate[Cha+11; McD+13] is an an embedded array language for computations for high-performance computing in Haskell. Computations on multi-dimensional, regular arrays are expressed in the form of parameterised collective operations, such as maps, reductions, and permutations. It is very similar to Repa[Kel+10] in the way computations are specified. The difference is in the way the computation gets evaluated. While Repa produces its result immediately, Accelerate collects the computation which can then be executed. This approach is necessary, as Accelerate isn't limited to be run on a CPU. The main back-end of accelerate is in fact based on CUDA.

### 2.2.1. Usage

To give an example of how to use Accelerate, lets look at the dot product of 2 vectors. This could easily implemented in Haskell like this.

```
dotp_list :: [Float] -> [Float] -> Float
dotp_list xs ys = foldl (+) 0 (zipWith (*) xs ys)
```

You can write nearly exactly the same function in Accelerate:

```
dotp :: Acc (Vector Float)
      -> Acc (Vector Float)
      -> Acc (Scalar Float)
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

The first difference is the use of a different datastructure. A `Vector` is an `Array` with one dimension. Similarly, a `Scalar` is an `Array` with 0 dimensions. These are then wrapped in the `Acc` type. `Acc a` is a computation, that yields an `a`.

The second difference is the `fold`. Unlike the `foldl` in the Haskell example, this `fold` doesn't specify the order of operations. This is important as it allows for an efficient implementation. In order of this to work, the operation passed to `fold` has to be associative.

If we want to call this function, we have to somehow produce values of type `Acc (Vector Float)`.

First the list is converted with `fromList`. To see what it does, lets first look at its type.

```
fromList :: (Elt e, Shape sh) => sh -> [e] -> Array sh e
```

In addition to the list of elements, `fromList` takes the shape of the resulting `Array`. Shapes can have an arbitrary amount of dimensions, but in this case it is only one. To construct a shape, there are 2 datatypes:

```
data Z = Z
data tail :: head = tail :: head
```

The head in this case represents the innermost dimension. The tail has to also be a shape.

With this, we get

```
type DIM0 = Z
type DIM1 = DIM0 :: Int
type DIM2 = DIM1 :: Int
...
```

and

```
type Scalar a = Array DIM0 a
type Vector a = Array DIM1 a
```

To lift the Arrays we get via `fromList` we use the function

```
use :: Arrays arrays => arrays -> Acc arrays
```

This is all we need to call `dotp`.

```
dotp' :: [Float] -> [Float] -> Acc (Scalar Float)
dotp' xs ys =
  let  xs' = use (fromList (Z ::length xs) xs)
      ys' = use (fromList (Z ::length xs) ys)
  in dotp xs' ys'
```



But we still only have the computation. To get the result out of the computation, we have to run it. After that we can convert the `Array` back to a list and extract the element.

```
dotp_list1 :: [Float] -> [Float] -> Float
dotp_list1 xs ys = head . toList $ run $ (dotp' xs ys)
```

### 2.2.2. Representation

Accelerate represents its arrays internally as regular unboxed one-dimensional arrays. In addition to this, the lengths of each dimension is stored. In case of a matrix, this would be height and width.

Rather than just simple types like `Int` or `Float`, Accelerate also supports tuples as elements. These cannot be easily stored in an unboxed array. Instead, arrays of tuples are stored as tuples of arrays.

```
Vector (Int, Float) ~ (Vector Int, Vector Float)
```

### 2.2.3. Skeletons

To execute any computation on a given architecture, the involved functions like `fold` or `map` have to be implemented. This is easy if the target is Haskell. The reason however that it is easy is the fact, that Haskell has support for higher-level functions like `fold` and `map`. In C, this can be emulated by using function pointers. This approach however is not very performant if the function is relatively simple as it involves multiple jumps and the manipulation of the call stack.

Accelerate uses skeletons instead. The idea is to write the function as you would normally, but leave blanks for the concrete types and passed function. When needed it can then be instantiated with the correct types and function. Figure 2.6 illustrates the idea using `map`.

Instead of the single array for input and output however, there will often be multiple. This is because a single array in Accelerate can be represented by multiple arrays internally.

### 2.2.4. Fusion

Consider the following example from earlier.

```
dotp xs ys = fold (+) 0 (zipWith (*) xs ys)
```

```
1 void map
2 (
3     TyOut *          d_out,
4     const TyIn *      d_in,
5     const int         length
6 ){
7     for (int i=0; i < length; ix += 1) {
8         d_out[i] = apply(d_in[i]);
9     }
10 }
```

Figure 2.6.: map Skeleton in C

This can be rewritten as

```
dotp xs ys =
  let zs = zipWith (*) xs ys
  in fold (+) 0 zs
```

First, the elements of the 2 arrays are combined pair-wise using `zipWith` and stored in a temporary array. As a second step, these elements are then combined into a `Scalar` using `fold`. Although this works perfectly well, it produces an intermediate array. This means additional writes and reads from memory. Considering that memory access is very costly this should better be avoided.

The solution is to delay the computation of the intermediate array. This means, that instead of computing the array in memory, it is done on-the-fly. This changes the construction of the skeletons somewhat. Instead of declaring the inputs directly, the arrays needed are passed in as an array environment. This is then accessed through a function `delayedLinearIndex`.

### 2.2.5. LLVM Backend

Apart from the CUDA-backend and the Interpreter, There are a number of incomplete backends for Accelerate. One of those that looks promising is the LLVM backend.[McD14] It supports both CPUs as well as NVIDIA GPUs. Unfortunately, it is not possible to “write once, run everywhere” with LLVM, as there are important differences between the architectures. That is why the Skeletons cannot be shared between the 2.

In this thesis, I will mainly work on the native backend, but the results should be transferable to the PTX backend as well.

uses gang workers.[Cha+07]

write about execution, but maybe wait for trevor to write stuff

## 3. Contributions

### 3.1. accelerate

While testing some of the skeletons, in particular the first stage of `scanl`, I had trouble with lost writes. These happened only if multiple threads wrote to adjacent memory locations. Normally this behaviour should be prevented by cache coherency protocols. These, however can be disabled if the memory involved is believed to be constant.

The Internal representation of Arrays in Accelerate uses a `UArray` to store it's data. This type of Array is uses a `ByteArray#`, which is assumed to be constant. Switching the representation from `UArray` to `StorableArray` fixed the problem.

### 3.2. llvm-general

While `llvm-general` offers very good bindings for

- Targetmachine (Optimization)
- fast-math

### 3.3. llvm-general-quote

When writing a compiler using LLVM in Haskell there is a good tutorial on how to do it at <http://www.stephendiehl.com/llvm/>. It uses *llvm-general* to interface with LLVM. The Idea followed in this tutorial is to use a monadic generator to produce the AST. The goal of monadic code generation is to use the state of the monad to store the instructions.

Figure 3.1 shows how to implement a simple for loop using monadic generators.

Unfortunately, this produces a lot of boilerplate code. We have to define the basic blocks manually and add the instructions one by one. This has some obvious drawbacks, as the code can get unreadable pretty quickly.

A more clean approach is to use a complete EDSL. The idea here is to use block structure to specify the individual elements. An implementation of this idea is *llvm-general-typed*. It actually goes further, as it also typechecks the produced code.

```

for :: Type                                -- type of the index
  -> Operand                              -- starting index
  -> (Operand -> CodeGen Operand)         -- loop test to keep going
  -> (Operand -> CodeGen Operand)         -- increment the index
  -> (Operand -> CodeGen ())              -- body of the loop
  -> CodeGen ()
for ti start test incr body = do
  loop <- newBlock "for.top"
  exit  <- newBlock "for.exit"

  -- entry test
  c    <- test start
  top  <- cbr c loop exit

  -- Main loop
  setBlock loop
  c_i <- freshName
  let i = local c_i

  body i

  i'    <- incr i
  c'    <- test i'
  bot   <- cbr c' loop exit
  _     <- phi loop c_i ti [(i',bot), (start,top)]

  setBlock exit

```

Figure 3.1.: Monadic generation of for loop

```

1  define i32 @foo(i32 %x) #0 {
2    entry:
3      br label %for.cond
4
5    for.cond:                                ; preds = %for.inc, %
      entry
6      %res.0 = phi i32 [ 0, %entry ], [ %add, %for.inc ]
7      %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
8      %cmp = icmp slt i32 %i.0, %x
9      br i1 %cmp, label %for.body, label %for.end
10
11   for.body:                                ; preds = %for.cond
12     %add = add nsw i32 %res.0, %x
13     %inc = add nsw i32 %i.0, 1
14     br label %for.cond
15
16   for.end:                                  ; preds = %for.cond
17     ret i32 %res.0
18 }

```

I propose a third approach using quasiquotation[Mai07]. The idea behind quasiquotation is, that you can define a DSL with arbitrary syntax, which you can then directly transform into Haskell data structures. This is done at compile-time, so you get the same type safety as writing the AST by hand.

Figure 3.3 shows how to implement a simple function in LLVM using quasiquotation. Compared to the other 2 solutions, this has already the advantage of being very close to the produced LLVM IR.

Without the ability to reference Haskell variables, this would be fairly useless in most cases. But quasiquotation allows for antiquotation as well. This means you can still reference arbitrary Haskell variables from within the quotation. Using this the following are equivalent:

- `[llinstr| add i64 %x, 1 |]`
- `let y = 1 in [llinstr| add i64 %x, $opr:y |]`

### 3.3.1. Control Structures

You still have to specify the  $\Phi$ -nodes by hand. This is fairly straight forward in a simple example, but can get more complicated very quickly.

The real goal is a language that “feels” like a high-level language, but can be trivially translated into LLVM. This means

- using LLVM instructions unmodified.
- introduce higher-level control structures like for, while and if-then-else.

```
1  [llg|
2  define i64 @foo(i64 %start, i64 %end) {
3      entry:
4          br label %for
5
6      for:
7          for i64 %i in %start to %end with i64 [0,%entry] as %x {
8              %y = add i64 %i, %x
9              ret i64 %y
10         }
11
12     exit:
13         ret i64 %y
14 }
15 |]
```

Figure 3.2.: For Loop using *llvm-general-quote*

Figure 3.2 shows a for loop using this approach. The loop-variable (%x in this case) is specified explicitly and can be referenced inside and after the for-loop. To update the %x, I overload the return statement to specify which value should be propagated. At the end of the for-loop the code automatically jumps to the next block. Figure 3.3 shows the produced LLVM code. This is clearly more readable.

### 3.3.2. SSA

The above approach unfortunately only works for some well-defined cases. With multiple loop-variables for example, it quickly becomes cluttered. On top of this, it relies on some “magic” to resolve the translation and legibility suffers.

The main reason why it is not possible to define a better syntax is that LLVM code has to be in SSA form. This means, that

1. every variable name can only be written to once.
2.  $\Phi$ -nodes are necessary where control flow merges.

To loosen this restriction, I implemented SSA recovery pass as part of the quasiquote. This means that I can now reassign variables inside the LLVM code. Using this, I was able to define a much more simple syntax for the `for`-loop. The placement of  $\Phi$ -nodes is not optimal, but redundant ones can be easily remove by LLVM’s `InstCombine` pass. Figure 3.4 and 3.5 show the quoted and the produced code after `InstCombine` respectively. The code produced by the for loop using SSA recovery is not identical to the code other code, but it is equivalent. In the new approach, the new loop counter is calculated at the end, whereas it was calculated in the loop head before.

```

1  define i64 @foo(i64 %start, i64 %end) {
2  entry:
3      br label %for
4
5  for:                                     ; preds = %for.body, %entry
6      %i = phi i64 [ %i.new, %for.body ], [ %start, %entry ]
7      %x = phi i64 [ %y, %for.body ], [ 0, %entry ]
8      %for.cond = icmp ule i64 %i, %end
9      %i.new = add nuw nsw i64 %i, 1
10     br i1 %for.cond, label %for.body, label %for.end
11
12 for.body:                               ; preds = %for
13     %y = add i64 %i, %x
14     br label %for
15
16 for.end:                                ; preds = %for
17     ret i64 %x
18 }

```

Figure 3.3.: Expanded For Loop

```

1  [llg]
2  define i64 @foo(i64 %start, i64 %end) {
3      entry:
4          %x = i64 0
5
6      for:
7          for i64 %i in %start to %end {
8              %x = add i64 %i, %x
9          }
10
11     exit:
12         ret i64 %x
13 }
14 []

```

Figure 3.4.: For Loop using *llvm-general-quote* and SSA

```
1  define i64 @foo(i64 %start, i64 %end) {  
2  entry:  
3      br label %for.head  
4  
5  for.head:                                ; preds = %n0, %entry  
6      %x.12 = phi i64 [ 0, %entry ], [ %x.6, %n0 ]  
7      %i.4 = phi i64 [ %start, %entry ], [ %i.9, %n0 ]  
8      %for.cond.3 = icmp slt i64 %i.4, %end  
9      br i1 %for.cond.3, label %n0, label %for.end  
10  
11 n0:                                       ; preds = %for.head  
12      %x.6 = add i64 %i.4, %x.12  
13      %i.9 = add nuw nsw i64 %i.4, 1  
14      br label %for.head  
15  
16 for.end:                                ; preds = %for.head  
17      ret i64 %x.12  
18 }
```

Figure 3.5.: Expanded For Loop (SSA)

### 3.3.3. syntax extensions

llvm-general-quote supports all llvm instructions. For better usage however, I added control structures like the for loop discussed earlier. These are the additions I made to the syntax:

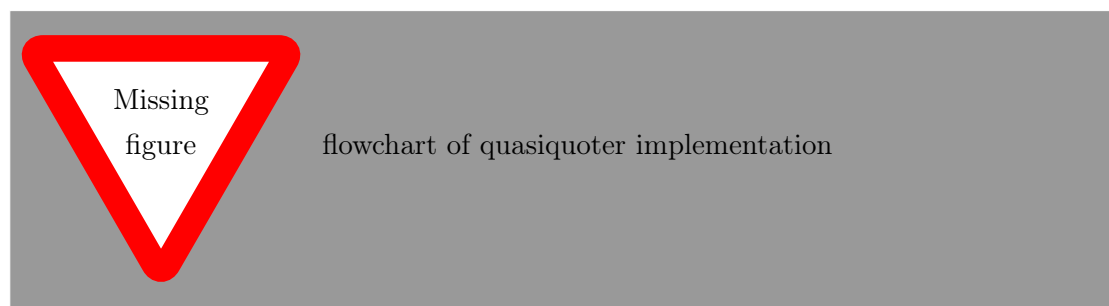
- direct assignment: `<var> = <ty> <val>`
- if: `if <cond> { <instructions> } [ else { <instructions> } ]`
- while: `while <cond> { <instructions> }`
- for: `for <ty> <var> in <vall> ( to | downto )<val2> { <instructions> }`



## 4. Implementation

### 4.1. Quasiquoter

The design of *llvm-general-quote* is inspired by *language-c-quote*, which is also used in the cuda implementation of Accelerate. I use “Happy” and *Alex*.



```
type Conversion a b = forall m. (CodeGenMonad m) => a -> TExpQ (m b)

class QQExp a b where
  qqExpM :: Conversion a b
  qqExp  :: a -> TExpQ b
  qqExp x = [|fst runState $$ (qqExpM x) ((0,M.empty) :: (Int,M.Map L.Name [L.Ope

class (Applicative m, Monad m) => CodeGenMonad m where
  newVariable    :: m L.Name
  exec           :: m () -> m [L.BasicBlock]
```

### 4.2. SSA

The standard method of producing SSA form in LLVM is to use stack allocated variables instead of registers to store values. The LLVM optimizer then uses the mem2reg pass to transform these into registers, adding the necessary phi-nodes in the process.

```
1 writeVariable(variable, block, value):
2     currentDef[variable][block] <- value
3
4 readVariable(variable, block):
5     if currentDef[variable] contains block:
6         # local value numbering
7         return currentDef[variable][block]
8     # global value numbering
9     return readVariableRecursive(variable, block)
```

Figure 4.1.: Implementation of local value numbering

Since one of the goals of the quasiquoter was to produce llvm-code that is as close to what the programmer wrote as possible, I decided against this approach. Instead I do the transformation myself. To this end, I provide a single function

```
toSSA :: [BasicBlock] -> [BasicBlock]
```

This is called with the list of all `BasicBlock`'s of a given function.

To produce SSA form, variables have to be versioned, so that it is clear which value it holds at each given time. There are a multitude of algorithms to do this. The one most widely used (including in LLVM) is Cytron et al.'s algorithm. It is however is rather involved as it relies on dominance frontiers. An easier approach was presented by Braun et al.

In this algorithm, the SSA is produced directly from the AST, without the need of extra analysis passes. To do this, the definitions of variables are tracked and updated as necessary. Without  $\Phi$ nodes, this is straight forward, as Figure 4.1 shows.

If the given variable has no definition in the current block, an empty  $\Phi$ -node is inserted and the local definition is set to the  $\Phi$ -node. Now the lookup is done recursively and the values are added to the  $\Phi$ -node. This extra step is necessary to detect loops correctly. I use a slightly different approach than presented in the paper. The original version tracks if the predecessors of a block are all processed. This is necessary since only then can the empty  $\Phi$ -nodes be filled. This approach makes sense if this is the norm. In case the input is an unordered list of blocks however, I can't make predictions about this. Instead I assume that no block is processed until all blocks are. Figure 4.2 shows the modified `readVariableRecursive`.

Like most efficient graph algorithms, this algorithm relies heavily on mutable data structures. Although Haskell - being purely functional - doesn't support these normally, there are multiple methods to implement them inside a pure Monad. The obvious choices are `State` and `ST`. `IO` also provides mutable state, but is not an option here, since there is no safe way of running it in a pure context. I use `ST`, since it natively provides an arbitrary number of mutable variables with direct access.

```

1 readVariableRecursive(variable, block):
2     if |block.preds| = 0:
3         # First block
4         val <- variable
5     else:
6         # Break potential cycles with operandless phi
7         val <- new Phi(block)
8         writeVariable(variable, block, val)
9     writeVariable(variable, block, val)
10    return val

```

Figure 4.2.: Implementation of global value numbering

To use this however, the BasicBlocks have to be in a mutable format.

```

type CFG s = [(Name, MutableBlock s)]

data MutableBlock s = MutableBlock {
    blockName :: Name,
    blockIncompletePhi :: STRef s (M.Map Name (MutableInstruction s)),
    blockPhi :: STRef s [MutableInstruction s],
    blockInstructions :: [MutableInstruction s],
    blockTerminator :: MutableTerminator s,
    blockPreds :: [Name],
    blockDefs :: STRef s (M.Map Name Name)
}

type MutableInstruction s = STRef s (Named Instruction)
type MutableTerminator s = STRef s (Named Terminator)

```

Although not everything is mutable in this representation, it is sufficient. After the conversion, the BasicBlocks can be processed one by one. First, the variables created by Instructions are adjusted. The same is done for usages. If there a variable is not defined in a given block and there are predecessors, an incomplete Phi instruction is added. Incomplete here means, that the values for every given predecessor are left undefined. When this is done, the variables referenced by existing Phi instructions are replaced by the new values.

Every time a recursive read is done, a new incomplete Phi instruction is added. These are handled last by reading the variable in all preceding blocks and replacing the undefined values. Since this involved reading however, new incomplete Phi instructions can be added in the process. To handle all of them, the function is executed until there are no incomplete Phi instructions.

Figure 4.3 shows the complete `toSSA` function.

```
toSSA :: [BasicBlock] -> [BasicBlock]
toSSA bbs = runST $ do
  cfg <- toCFG $ bbs
  ctr <- newSTRef 1

  -- process all Instructions
  mapM_ (blockToSSAPre ctr) (map snd cfg)
  -- replace names in Phis with correct references
  mapM_ (blockToSSAPhi ctr cfg) (map snd cfg)
  -- replace names in newly added Phis with correct references
  handleIncompletePhis ctr cfg

  fromCFG cfg
```

Figure 4.3.: toSSA

### 4.3. Control Structures

When implementing the control structures, I had to decide on what level I wanted to introduce it. In a traditional procedural language like C, they would sit alongside the other expressions like assignments or function calls. This would correspond to the instructions in LLVM. For this to be possible however, I need to be able to extend them into just a sequence of instructions. In case of an if-then-else this is still kind of possible using select to get the result value. The loop structures however are not possible, as they require an arbitrary number of jumps.

#### 4.3.1. first version (direct)

The solution is to have the control structures on the level of basic blocks.

- cumbersome syntax
- limited functionality

#### 4.3.2. second version (mutable variables)

- simpler syntax
- very flexible in its use
- simpler implementation

## 5. Skeletons

### 5.1. map

### 5.2. fold

### 5.3. scan

[LF80] My plan is the following:

```
1  void scan(double* in, double* out, double* tmp, unsigned length,
    unsigned start, unsigned end, unsigned tid) {
2      double acc = 0;
3      if (end==length) {
4          tmp[0] = 0;
5      } else {
6          for (unsigned i=start; i<end; i++) {
7              acc += in[i];
8          }
9          tmp[tid+1] = acc;
10     }
11     printf("block");
12     acc = tmp[tid];
13     for (unsigned j=start; j<end; j++) {
14         acc += in[j];
15         out[j] = acc;
16     }
17 }
18
19 void scanAlt(double* in, double* out, unsigned length, unsigned start,
    unsigned end) {
20     double acc = 0;
21     for (unsigned j=start; j<end; j++) {
22         acc += in[j];
23         out[j] = acc;
24     }
25     printf("block");
26     double add=0;
27     if (start>0) {
28         add = out[start-1];
29     }
30     for (unsigned i=start; i<end; i++) {
31         out[i] += add;
```

```
32     }  
33 }
```

## 5.4. stencil

## 6. Conclusion

### 6.1. Related Work





## A. Listings



# References

- [AWZ88] ALPERN, Bowen ; WEGMAN, Mark N ; ZADECK, F Kenneth: Detecting equality of variables in programs. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1988, pp. 1–11. URL: <https://courses.cs.washington.edu/courses/cse501/04wi/papers/alpern-popl88.pdf>
- [Bra+13] BRAUN, Matthias et al.: Simple and Efficient Construction of Static Single Assignment Form. In: JHALA, Ranjit ; BOSSCHERE, Koen (eds.): *Compiler Construction*. Vol. 7791. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 102–122. DOI: 10.1007/978-3-642-37051-9\_6. URL: [http://dx.doi.org/10.1007/978-3-642-37051-9\\_6](http://dx.doi.org/10.1007/978-3-642-37051-9_6)
- [Cha+11] CHAKRAVARTY, Manuel MT et al.: Accelerating Haskell array codes with multicore GPUs. In: *Proceedings of the sixth workshop on Declarative aspects of multicore programming*. ACM. 2011, pp. 3–14. URL: <http://morri.web.cse.unsw.edu.au/~chak/papers/acc-cuda.pdf>
- [Cha+07] CHAKRAVARTY, Manuel MT et al.: Data Parallel Haskell: a status report. In: *Proceedings of the 2007 workshop on Declarative aspects of multicore programming*. ACM. 2007, pp. 10–18. URL: <http://www.cse.unsw.edu.au/~chak/papers/data-parallel-haskell.pdf>
- [Cyt+91] CYTRON, Ron et al.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. In: *ACM Transactions on Programming Languages and Systems* 13 (Oct. 1991) Nr. 4, pp. 451–490. URL: <http://doi.acm.org/10.1145/115372.115320>
- [Die14] DIEHL, Stephen: *Implementing a JIT Compiled Language with Haskell and LLVM*. Jan. 2014. URL: <http://www.stephendiehl.com/llvm/>
- [DM] DORNAN, Chris ; MARLOW, Simon: *Alex: A lexical analyser generator for Haskell*. URL: <http://www.haskell.org/alex/>
- [GM95] GILL, Andy ; MARLOW, Simon: Happy: the parser generator for Haskell. In: *University of Glasgow* (1995)
- [Hol14] HOLTZ, Timo von: *llvm-general-quote*. 2014. URL: <https://github.com/tvh/language-llvm-quote>

- [How14] HOWELL, Nathan: *llvm-general-typed*. 2014. URL: <https://github.com/alphaHeavy/llvm-general-typed>
- [Kel+10] KELLER, Gabriele et al.: Regular, shape-polymorphic, parallel arrays in Haskell. In: *ACM Sigplan Notices*. Vol. 45. 9. ACM. 2010, pp. 261–272. URL: <http://www.cse.unsw.edu.au/~chak/papers/repa.pdf>
- [LF80] LADNER, Richard E ; FISCHER, Michael J: Parallel prefix computation. In: *Journal of the ACM (JACM)* 27 (1980) Nr. 4, pp. 831–838. URL: <https://engr.smu.edu/~seidel/courses/cse8351/papers/LadnerFischer80.pdf>
- [Lat02] LATNER, Chris Arthur: *LLVM: An infrastructure for multi-stage optimization*. MA thesis. University of Illinois, 2002. URL: <https://llvm.org/svn/llvm-project/www-pubs/trunk/2002-12-LattnerMSThesis-book.pdf>
- [LLV14] LLVM PROJECT: *Auto-Vectorization in LLVM*. [Online; accessed 1-July-2014]. 2014. URL: <http://llvm.org/docs/Vectorizers.html>
- [LLV] LLVM PROJECT: *LLVM Language Reference Manual*. [Online; accessed 1-July-2014]. URL: <http://llvm.org/docs/LangRef.html>
- [Mai07] MAINLAND, Geoffrey: Why it’s nice to be quoted: quasiquoting for haskell. In: *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*. ACM. 2007, pp. 73–82. URL: <https://www.cs.drexel.edu/~mainland/publications/mainland07quasiquoting.pdf>
- [MC] MAINLAND, Geoffrey ; CHAKRAVARTY, Manuel MT: *language-c-quote*. URL: <http://hackage.haskell.org/package/language-c-quote>
- [McD14] McDONELL, Trevor L: *accelerate-llvm*. 2014. URL: <https://github.com/AccelerateHS/accelerate-llvm>
- [McD+13] McDONELL, Trevor L et al.: Optimising Purely Functional GPU Programs. In: *Proceedings of the The 18th ACM SIGPLAN International Conference on Functional Programming*. 2013. URL: <http://cs3141.web.cse.unsw.edu.au/~chak/papers/acc-optim.pdf>
- [RWZ88] ROSEN, Barry K ; WEGMAN, Mark N ; ZADECK, F Kenneth: Global value numbers and redundant computations. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1988, pp. 12–27. URL: <http://www.cs.wustl.edu/~cytron/cs531/Resources/Papers/valnum.pdf>
- [Sca13] SCARLET, Benjamin: *llvm-general*. 2013. URL: <http://hackage.haskell.org/package/llvm-general>