Master Thesis

Generating LLVM IR using Template Meta Programming in Haskell

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, August 29, 2014

Selbstständigkeitserklärung

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

August 29, 2014

Timo von Holtz

Abstract

Implementing parallel computations is traditionally done in low level languages like 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

Contents

List of Figures

List of Tables

1 Introduction

Processing large amounts of data is becoming more and more important. To meet this demand, there have been a number of approaches. Most of these approaches involve multiple processing cores in some form. This comes at a cost however, as it is no longer possible to use all the power given just by implementing a sequential algorithm.

Many computing-intense calculations can be expressed as operations on arrays. This class of problems is addressed by Accelerate[Cha+11; McD+13]. Accelerate is an an embedded array language for computations for high-performance computing in Haskell.

The way an algorithm is expressed in Accelerate is using a set of combinator functions like **`map`** or **`fold`**. To give an example, lets look at the dot product of 2 vectors.

$$\begin{pmatrix} x_0 \\ \dots \\ x_{n-1} \end{pmatrix} \times \begin{pmatrix} y_0 \\ \dots \\ y_{n-1} \end{pmatrix} = \sum_{i=0}^{n-1} x_i * y_i$$

This can be trivially implemented in Haskell using **`foldl'`**[1] and **`zipWith`**.

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

This code is purely sequential however and is therefore limited to only one CPU. Accelerate allows you to write this computation in nearly exactly the same way.

[1] The implementation of **`foldl`** in Haskell is such, that it will produce large intermediate chunks. **`foldl'`** is a strict variant, so this doesn't happen.

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

In order to run this computation in parallel, there has to be an implementation leveraging the possible parallelism. The implementation presented in the original Accelerate work targets NVIDIA GPUs using CUDA, as those offered the most performance, while being traditionally hard to program. This works great in a specialized setting where there is a powerful GPGPU available. If there is no GPGPU available to run this computation however, this implementation won't work. To use Accelerate in a more general setting, there has been early work on an alternative backend to Accelerate using LLVM.[McD14] This work includes both a CPU and a GPU backend. As this is very early work, there are still missing pieces to support the full feature set of Accelerate. One of these pieces is the code generation for the different combinator function.

In this thesis I will develop a novel approach to code generation for LLVM. I will use this to implement the missing pieces in the code generation of the native LLVM backend.

2 Technologies

To implement the code generation for LLVM, I have to touch a lot of different technologies. In this chapter I will give an overview of the different tools and technologies I used.

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. This 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 an idea of how this looks in practice, let's look at an example. Figure 2.1 shows a simple C function to sum up the elements of an array and the corresponding LLVM.

```c
int sum(int* a, int length) {
  int x = 0;
  for (int i=0;i<length;i++) {
    x += a[i];
  }
  return x;
}
```

(a) C

```
define i32 @sum(i32* nocapture readonly %a, i32 %length) {
entry:
  %cmp4 = icmp sgt i32 %length, 0
  br i1 %cmp4, label %for.body, label %for.end

for.body:                                         ; preds = %entry, %for.body
  %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ],
                        [ 0, %entry ]
  %x.05 = phi i32 [ %add, %for.body ], [ 0, %entry ]
  %arrayidx = getelementptr inbounds i32* %a, i64 %indvars.iv
  %0 = load i32* %arrayidx, align 4
  %add = add nsw i32 %0, %x.05
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv.next to i32
  %exitcond = icmp eq i32 %lftr.wideiv, %length
  br i1 %exitcond, label %for.end, label %for.body

for.end:                                          ; preds = %for.body, %entry
  %x.0.lcssa = phi i32 [ 0, %entry ], [ %add, %for.body ]
  ret i32 %x.0.lcssa
}
```

(b) LLVM

Figure 2.1: Sum of an array in C and in LLVM

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**, **mul**, ..., but also **call**. Terminators can either be used to jump to another block (branch) or return to the calling function.

To handle dynamic control flow, there are Φ-nodes. These specify the the value of a new variable depending on what the last block was. The Φ-nodes in the SSA are represented with **phi**-instructions. These have to preceed every other instruction in a given basic block.

Types

The LLVM type system is one of the most important features of the intermediate representation. Having this type information 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, lets 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 the odd and even elements 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 2.2 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[1].

[1] `http://llvm.org/docs/Vectorizers.html`

```
define i32 @sum(i32* nocapture readonly %a, i32 %length) {
entry:
  %cmp4 = icmp sgt i32 %length, 0
  br i1 %cmp4, label %for.body.preheader, label %for.end

for.body.preheader:                   ; preds = %entry
  %0 = add i32 %length, -1
  %1 = zext i32 %0 to i64
  %2 = add i64 %1, 1
  %end.idx = add i64 %1, 1
  %n.vec = and i64 %2, 8589934590
  %cmp.zero = icmp eq i64 %n.vec, 0
  br i1 %cmp.zero, label %middle.block, label %vector.body

vector.body:                          ; preds = %for.body.preheader, %vector.body
  %index = phi i64 [ %index.next, %vector.body ], [ 0, %for.body.preheader ]
  %vec.phi = phi <2 x i32> [ %5, %vector.body ],
                           [ zeroinitializer, %for.body.preheader ]
  %3 = getelementptr inbounds i32* %a, i64 %index
  %4 = bitcast i32* %3 to <2 x i32>*
  %wide.load = load <2 x i32>* %4, align 4
  %5 = add nsw <2 x i32> %wide.load, %vec.phi
  %index.next = add i64 %index, 2
  %6 = icmp eq i64 %index.next, %n.vec
  br i1 %6, label %middle.block, label %vector.body

middle.block:                         ; preds = %vector.body, %for.body.preheader
  %resume.val = phi i64 [ 0, %for.body.preheader ], [ %n.vec, %vector.body ]
  %rdx.vec.exit.phi = phi <2 x i32> [ zeroinitializer, %for.body.preheader ],
                                    [ %5, %vector.body ]
  %rdx.shuf = shufflevector <2 x i32> %rdx.vec.exit.phi, <2 x i32> undef,
                                                         <2 x i32> <i32 1, i32 undef>
  %bin.rdx = add <2 x i32> %rdx.vec.exit.phi, %rdx.shuf
  %7 = extractelement <2 x i32> %bin.rdx, i32 0
  %cmp.n = icmp eq i64 %end.idx, %resume.val
  br i1 %cmp.n, label %for.end, label %for.body

for.body:                             ; preds = %middle.block, %for.body
  %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ %resume.val, %middle.block ]
  %x.05 = phi i32 [ %add, %for.body ], [ %7, %middle.block ]
  %arrayidx = getelementptr inbounds i32* %a, i64 %indvars.iv
  %8 = load i32* %arrayidx, align 4
  %add = add nsw i32 %8, %x.05
  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv.next to i32
  %exitcond = icmp eq i32 %lftr.wideiv, %length
  br i1 %exitcond, label %for.end, label %for.body

for.end:                              ; preds = %for.body, %middle.block, %entry
  %x.0.lcssa = phi i32 [ 0, %entry ], [ %add, %for.body ], [ %7, %middle.block ]
  ret i32 %x.0.lcssa
}
```

Figure 2.2: Vectorized Sum

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 however. To vectorize the code regardless, LLVM has the notion of fast-math-flags[LLV]. These tell the optimizer to assume certain properties that aren't true in general.

Available flags are:

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.

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, Φ-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).

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

The SLP vectorizer may combine these into vector operations. Figure 2.3 shows the corresponding LLVM before and after SLP vectorization.

2.1.3 llvm-general

To use LLVM from Haskell, there are multiple options. 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. To use the optimization, there are

```
withPassManager :: PassSetSpec -> (PassManager -> IO a) -> IO a
runPassManager :: PassManager -> Module -> IO Bool
```

A `PassSetSpec` specifies which passes should be run. There are 2 different options to define a `PassSetSpec` that reflect different usecases. 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.

```
define void @foo(double %a1, double %a2,
                 double %b1, double %b2, double* %A)
{
  %add = fadd double %a1, %b1
  %mul = fmul double %add, %a1
  %div = fdiv double %mul, %b1
  %mul1 = fmul double %b1, 5.000000e+01
  %div2 = fdiv double %mul1, %a1
  %add3 = fadd double %div, %div2
  store double %add3, double* %A, align 8
  %add4 = fadd double %a2, %b2
  %mul5 = fmul double %add4, %a2
  %div6 = fdiv double %mul5, %b2
  %mul7 = fmul double %b2, 5.000000e+01
  %div8 = fdiv double %mul7, %a2
  %add9 = fadd double %div6, %div8
  %arrayidx10 = getelementptr inbounds double* %A, i64 1
  store double %add9, double* %arrayidx10, align 8
  ret void
}
```

(a) `foo` without SLP vectorization

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

(b) `foo` with SLP vectorization

Figure 2.3: Example for SLP vectorization

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 necessaray, 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 vector product I used in the Introduction.

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

If this was implemented in regular Haskell, it would look like this.

```
dotp_list :: [Float] -> [Float] -> Float
dotp_list xs ys = foldl (+) 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`. In this sense `Acc` forms a monad, but is limited to array types.

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. To maintain a defined result, the operation passed to `fold` has to be associative.

To call `dotp`, the source data first has to be converted to the right format. To do this, the list is first converted using `fromList`. To see what it does, lets 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` to a computation 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 2.4 illustrates the idea using **map**.

```
void map
(
  TyOut *           d_out,
  const TyIn *      d_in,
  const int         length
){
  for (int i=0; i < length; ix += 1) {
    d_out[i] = apply(d_in[i]);
  }
}
```

Figure 2.4: **map** Skeleton in C

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

2.2.4 Fusion

Consider the following example from earlier.

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

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 uses llvm-general to bind to the LLVM API. It supports both CPUs as well as NVIDIA GPUs through PTX. In contrast to the CUDA backend, this has the advantage that no external program has to be called for compilation of the kernels. This is also true for the CPU side. Unfortunately it is not quite possible to "write once, run everywhere" with LLVM. Although the instructions are exactly the same between architectures, the supplied libraries are not. For this reason the Skeletons have to be written for the 2 backends separately.

To run a computation in the LLVM native backend, the skeletons are first optimized and compiled to a callable function. Similar to Repa, it uses gang workers[Cha+07] for execution. These are multiple threads – typically as many as there are CPU cores available. They wait on an `MVar` for IO actions to perform. At first, the load is split evenly between the workers. If a thread finishes early however, it looks for more work. If it finds that another thread has excess work available, it will try to steal parts of that work (usually half). This process is called work stealing. On a shared memory architecture this has very little overhead as no data has to be physically copied.

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

2.3 Template Haskell

Template Haskell is an extension to Haskell that allows you to do type-safe compile-time meta-programming, with Haskell both as the manipulating language and the language being manipulated. In GHC versions prior to 7.8, one would get a Template Haskell expression using quasiquoters like this.

```
foo :: Q Exp
foo = [|Just 5|]
```

It is not clear which type of expression it is. Another nasty side-effect of this is that the expression is not checked to be consistent internally. That leads to something like this being legal.

```
bar :: Q Exp
bar = [|case 1 of
          Nothing -> 42
          True    -> ()|]
```

Apart from being syntactically correct, this makes no sense at all. With GHC 7.8, a new form of quoting Haskell code was introduced.

```
foo :: Q (TExp (Maybe Int))
foo = [||Just 5||]
```

The code is now typechecked along with the rest of the code in the module. This means it is no longer possible to easily produce nearly untraceable type errors with Template Haskell. The structure of the expressions remains unchanged however.

```
newtype TExp a = TExp { unType :: Exp }
```

This means it is still possible to use typed expressions in places where untyped expressions are expected. The reverse is also true, but type safety is lost in the process.

```
unsafeTExpCoerce :: Q Exp -> Q (TExp a)
```

To use the defined expression, it can then be spliced in using `$$(..)`.

This, however, is not possible everywhere. The following for example doesn't compile.

```
bar :: Int
bar = let x = [||1||]
      in $$(x)
```

And produces this error message:

```
GHC stage restriction:
  'x' is used in a top-level splice or annotation,
  and must be imported, not defined locally
In the splice: $$x
```

Template Haskell code has to be executable on compile-time. This means that in a splice, all expressions must be known. As a consequence, they can't be defined locally or passed in as function arguments. Not having this restriction would lead to Haskell not being type safe.

3 Contributions

Before we look at the intricate details I want to summarize the contributions I made. First is of course my approach to generate LLVM code via template meta programming. While working with accelerate-llvm, I discovered a few additional obstacles however, which I will also present here.

3.1 llvm-general-quote

When writing a compiler using LLVM in Haskell there is a good tutorial on how to do it at[1]. 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.

[1] `http://www.stephendiehl.com/llvm/`

```
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

```
foo :: Global
foo = [llg|
  define i32 @foo(i32 %x) #0 {
  entry:
    br label %for.cond

  for.cond:
    %res.0 = phi i32 [ 0, %entry ], [ %add, %for.inc ]
    %i.0 = phi i32 [ 0, %entry ], [ %inc, %for.inc ]
    %cmp = icmp slt i32 %i.0, %x
    br i1 %cmp, label %for.body, label %for.end

  for.body:
    %add = add nsw i32 %res.0, %x
    %inc = add nsw i32 %i.0, 1
    br label %for.cond

  for.end:
    ret i32 %res.0
  }
  |]
```

Figure 3.2: Simple example of quasiquoted LLVM

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 you would writing the AST by hand.

Figure 3.2 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 |]`

3.1.1 Control Structures

You still have to specify the Φ-nodes by hand in order to support dynamic control flow. 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.

Figure 3.3a 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 3.3b shows the produced LLVM code. This is clearly more readable.

3.1.2 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. Φ-nodes are necessary where control flow merges.

```
[llg|
define i64 @foo(i64 %start, i64 %end) {
  entry:
    br label %for

  for:
    for i64 %i in %start to %end with i64 [0,%entry] as %x {
        %y = add i64 %i, %x
        ret i64 %y
    }

  exit:
    ret i64 %y
}
|]
```

(a) Quoted for-loop

```
define i64 @foo(i64 %start, i64 %end) {
entry:
  br label %for

for:                                ; preds = %for.body, %entry
  %i = phi i64 [ %i.new, %for.body ], [ %start, %entry ]
  %x = phi i64 [ %y, %for.body ], [ 0, %entry ]
  %for.cond = icmp ule i64 %i, %end
  %i.new = add nuw nsw i64 %i, 1
  br i1 %for.cond, label %for.body, label %for.end

for.body:                           ; preds = %for
  %y = add i64 %i, %x
  br label %for

for.end:                            ; preds = %for
  ret i64 %x
}
```

(b) Expanded for-loop

Figure 3.3: For Loop using draft of *llvm-general-quote*

To loosen this restriction, I implemented SSA recovery pass as part of the quasiquoter. 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 Φ-nodes is not optimal, but redundant ones can be easily remove by LLVM's InstCombine pass.

Figure 3.4 shows the quoted and the produced code after InstCombine. The code produced by the for loop using SSA recovery is not identical to the 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.

3.1.3 Antiquotation

An important part of every quasiquoter is the ability to reference other variables in scope. The available antiquotations are:

- $dl: `DataLayout`
- $tt: target triple (**`String`**)
- $def: `Definition`
- $defs: `[Definition]`
- $bb: `BasicBlock`
- $bbs: `[BasicBlock]`
- $instr: `Named Instruction`
- $instrs: `[Named Instruction]`
- $type: `Type`
- $opr: `Operand`
- $const: `Constant`
- $id: `Name` (local)
- $gid: `Name` (global)
- $param: `Parameter`
- $params: `[Parameter]`

```
[llg|
define i64 @foo(i64 %start, i64 %end) {
  entry:
    %x = i64 0

  for:
    for i64 %i in %start to %end {
        %x = add i64 %i, %x
    }

  exit:
    ret i64 %x
}
|]
```

(a) Quoted code

```
define i64 @foo(i64 %start, i64 %end) {
entry:
  br label %for.head

for.head:                         ; preds = %n0, %entry
  %x.12 = phi i64 [ 0, %entry ], [ %x.6, %n0 ]
  %i.4 = phi i64 [ %start, %entry ], [ %i.9, %n0 ]
  %for.cond.3 = icmp slt i64 %i.4, %end
  br i1 %for.cond.3, label %n0, label %for.end

n0:                               ; preds = %for.head
  %x.6 = add i64 %i.4, %x.12
  %i.9 = add nuw nsw i64 %i.4, 1
  br label %for.head

for.end:                          ; preds = %for.head
  ret i64 %x.12
}
```

(b) Expanded loop

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

3.1.4 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 <val1> ( to | downto ) <val2> { <instructions> }`

3.2 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.

3.3 llvm-general

llvm-general offers a nice set of bindings to the LLVM API. It is not feature-complete however. One area where I could make an improvement on this is the optimization. The sole difference between the `PassSetSpect` and `CuratedPassSetSpec` should be the way they define passes. The `CuratedPassSetSpec` however was lacking important

fields for data layout and target machine.[2] Also, they were mostly ignored when supplied to the `PassSetSpec`.[3] In the process, I also added support for loop and slp vectorization to `CuratedPassSetSpec`.

Another issue was the lack of support for fast-math flags.[4] They are now supported with the following datatype:

```
data FastMathFlags
  = NoFastMathFlags
  | UnsafeAlgebra
  | FastMathFlags {
      noNaNs :: Bool,
      noInfs :: Bool,
      noSignedZeros :: Bool,
      allowReciprocal :: Bool }
```

[2] https://github.com/bscarlet/llvm-general/pull/101
[3] https://github.com/bscarlet/llvm-general/issues/91
[4] https://github.com/bscarlet/llvm-general/issues/90

4 Implementation

In the previous chapter I presented the extent of my work. In this chapter I will focus on the details of the Implementation.

4.1 Quasiquoter

A quasiquoter basically works like a mini compiler. You get a string as input and have to generate Haskell code out of that. In it's simplest form this would be just a parser.

It is not limited to that however. All you need to define for a quasiquoter is a function

```
quoteExp :: String -> Q Exp
```

The `Q Exp` in this case is a Template Haskell Expression. Since it uses Template Haskell to construct expressions, it is possible to reference variables in scope using antiquotation. Figure 4.1 shows a the design of a typical quasiquoter.

The design of *llvm-general-quote*, the quasiquoter I defined for LLVM, is inspired by *language-c-quote*, which is also used in the cuda implementation of Accelerate. This means I use *Alex* to specify the lexer and "Happy" for the parser. The goal of this quasiquoter is to generate an AST as defined by llvm-general-pure. If I want to do more than just parse complete LLVM code, I can't use this as a target for the parser. This means it is necessary to copy almost the whole AST and add additional constructors for antiquotation etc. This step wasn't necessary in *language-c-quote* since it's target datastructure still has constructors for antiquotation.

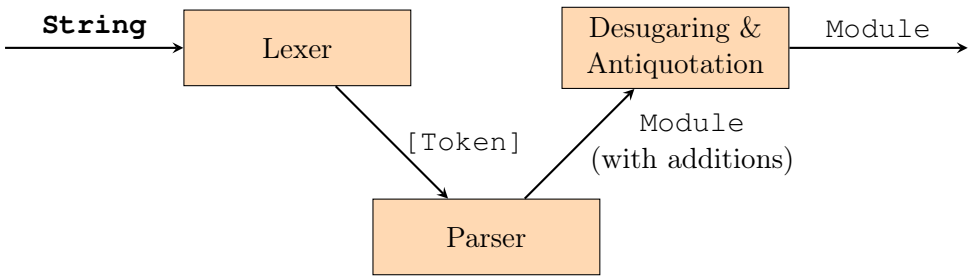


Figure 4.1: flowchart quasiquoter

This is where the 2 quasiquoters differ significantly in their construction. *language-c-quote* assumes that constructors stay the same, as long as there isn't a different case available. It does this by using the function

```
dataToExpQ :: Data a
           => (forall b. Data b => b -> Maybe (Q Exp))
           -> a
           -> Q Exp
```

This functions converts anything that is of typeclass `Data` into a Template Haskell expression. In addition to this however, it also takes an extra function to handle exceptions. This gets executed on every subterm recursively. If it returns **`Nothing`**, then the term gets translated into an expression directly. It it returns **`Just exp`**, then **`exp`** is used instead. The transformation function is then glued together as

```
qqExp :: Typeable a => a -> Maybe (Q Exp)
qqExp = const Nothing  `extQ` qqStringE
                       `extQ` qqIdE
...
```

I could have used this approach, but since I didn't use the same datatype for input and output, I opted for a different solution involving a typeclass.

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

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

This typeclass provides a projection between source and target datastructure. The `CodegenMonad` is important mostly to supply new names to unnamed basic blocks. It also provides an iterface to antiquote a list of basic blocks produced by monadic code generation. This is necessary to interface with existing code in accelerate-llvm.

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

The `Applicative` context is not necessary since **Monad** is strictly more powerful than `Applicative`.[1] I decided to include it anyway, as it allows for a more concise syntax im my Implementation.

Using a typeclass for the conversion instead of a single function like in language-c-quote has multiple advantages. When there is a case missing, it will complain when the quoter is compiled instead of when it is used. To implement this approach however, the resulting expressions have to be properly typed. This was not possible before GHC 7.8.

Here is a simple example of an instance declaration of `QQExp`.

```
instance QQExp A.Module L.Module where
  qqExpM (A.Module n dl tt ds) =
    [||L.Module <$> $$(qqExpM n) <*> $$(qqExpM dl)
                <*> $$(qqExpM tt) <*> $$(qqExpM ds)||]
```

[1] With GHC 7.10, `Applicative` will become a superclass of **Monad**. In the meantime, all instances of **Monad** should be adapted to also provide an instance of `Applicative`. Details can be found at `http://www.haskell.org/haskellwiki/Functor-Applicative-Monad_Proposal`.

In this case the structure of both the source and target constructors are the same.[2] Antiquotations have to be handled a little different. Since there can't be any information on the antiquoted expression, it has no specific type. To use it in a typed expression anyway, it has to be coerced.

```
instance QQExp A.Operand L.Operand where
  qqExpM (A.AntiOperand s) =
    [||$$(unsafeTExpCoerce $ antiVarE s)||]
```

The typed expression is first produced and then spliced it into a quoted expression. I could have omitted this roundtrip and used the coerced value direcly.

In some cases antiquotation is still a bit cumbersome. One of these cases is the use of numerical constants. To antiquote those, it would be necessary to first construct a `Constant`. I decided it would be nicer if it was also possible to use ordinnary Haskell values like `Word32` and **`Float`** directly. To this end, I created a class

```
class ToConstant a where
  toConstant :: a -> L.Constant
```

With this it is possible to just write the following:

```
foo :: Instruction
foo = let i = 5::Word32
      in [lli|add i32 %x, $const:i|]
```

To integrate this into `QQExp`, the intuitive way would be to write something like the following.

```
instance QQExp A.Constant L.Constant where
  qqExpM (A.AntiConstant s) =
    [||$$(unsafeTExpCoerce $ antiVarE s)
         >>= (return . toConstant)||]
```

[2] Instances like this could be generated using a Template Haskell function. I decided against using Template Haskell to generate the functions here, since this would make the code harder to debug.

Unfortunately this leads to a type error.

```
src/LLVM/General/Quote/Base.hs:681:54:
    Could not deduce (ToConstant a0) arising from
                                          a use of 'toConstant'
    from the context (CodeGenMonad m)
      bound by the type signature for
         qqConstantE :: CodeGenMonad m => A.Constant
                     -> TExpQ (m L.Constant)
      at src/LLVM/General/Quote/Base.hs:657:16-47
    The type variable 'a0' is ambiguous
...
```

The reason this happens is, that GHC can't know the correct type, as it will only be instantiated when the quasiquter is used. There are multiple workarounds for this problem. One is to define a correct type arbitrarily.

```
qqConstantE (A.AntiConstant s) =
  [||$$(unsafeTExpCoerce $ antiVarE s
          :: forall m. CodeGenMonad m => m Double)
       >>= return . toConstant||]
```

This obviously gets rid of the previous error. One would think however, that this will limit the antiquotation to only support **Double**. This is however not the case, as there is no real information about the types inside the Template Haskell Expression. To make this more clear, I opted instead to just use the old-style Template Haskell.

```
qqConstantE (A.AntiConstant s) =
  unsafeTExpCoerce [|$(antiVarE s) >>= (return . toConstant)|]
```

4.2 Control Structures

When implementing the control structures, I had to decide on what level I wanted to introduce them. 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 would kind of be possible using select to get the result value. This doesn't really work however, as side effects are still executed. Loop structures are an even bigger problem, as the number of iterations can be arbitrary.

4.2.1 Direct Approach

The solution is to have the control structures on the level of basic blocks. This way it is possible to just produce multiple basic blocks. For something to behave like a basic block, it must have all the elements of a basic block. In llvm-general, a basic block is represented as

```
data BasicBlock =
  BasicBlock Name [Named Instruction] (Named Terminator)
```

It consists of a name, a list of instructions and a terminator. The name is used as a target label for jumps. The terminator can either be a jump to a different block or a return statement. The instructions are just a stream without any jumps, but can also contain function calls.

Lets look at a classic for-loop. It starts by initializing a counter. Then the counter is checked against a maximum. If this is not yet reached, the loop body gets executed and the counter incremented. If it is, then the loop exits and the next expression is evaluated. Figure 4.2 shows the process.

To translate this into LLVM, I need a label, the name of the counter, it's minimum and maximum value and the loop body. In a normal language these would be sufficient. LLVM doesn't have mutable variables, however. This means that any state, like an accumulator, needs to be explicitly defined. The syntax I chose for this is the following:

```
for <ty1> <var1> in <val1> to <val2>
                 with <ty2> <values> as <var2>,
```

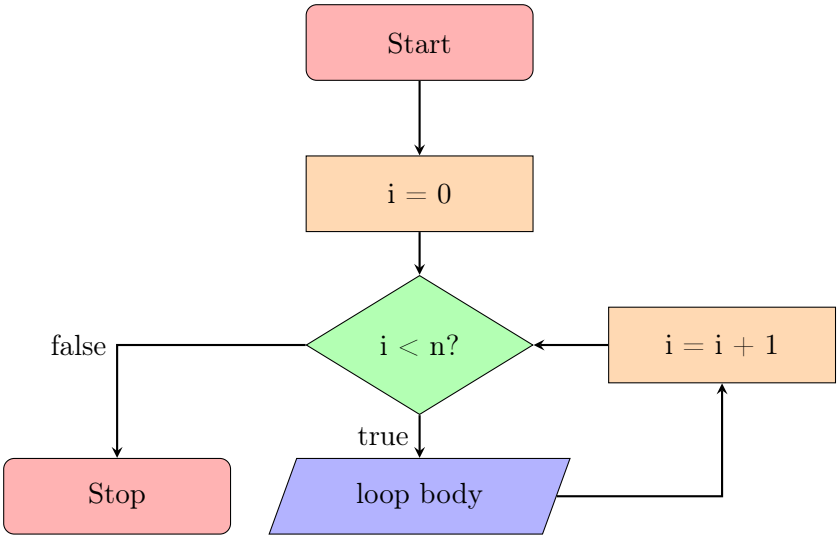


Figure 4.2: flowchart for-loop

```
label <jumptarget> { <loop body> }
```

The values here work the same way as with **phi**-instructions. It is often necessary to nest loops. To allow for this, the loop body can consist of multiple basic blocks. This way, it works much like a function body, with the loop counter and the accumulator as arguments. Going further with this analogy, I reuse the return statement to indicate the end of the loop. The value returned is used as the new value of the accumulator.

A simple function summing up the values from `%m` to `%n-1` would look like the following.

```
i32 foo(i32 %m, i32 %n) {
  entry:
    br label %for

  for:
    for i32 %i in %m to %n with i32 [ 0, %entry ] as %j,
                                                label %end {
        %k = add i32 %i, %j
```

```
        ret i32 %k
    }

  end:
    ret i32 %j
}
```

When implementing this, there are a few things that make matters non-trivial. Since LLVM doesn't have a concept of mutable variables, it is necessary to introduce **phi** instructions manually. The values that are modified are the accumulator and the loop counter (`%j` and `%i` in this example). Each **phi** instruction needs one value for each incoming block. The incoming blocks can be deduced by combining the existing list of incoming values with the return values of block within the loop. This naturally gives a complete list for the accumulator value. An additional **phi** instruction is then inserted to handle the loop counter.

This transformation is relatively straight forward if implemented as a regular Haskell function. Working with quasiquoters though, It all has to be implemented in a Template Haskell Expression. This means that it is sometimes necessary to move code around just so that it compiles, although the types are correct. The reason for this is the stage restriction of Template Haskell. A value cannot be spliced into an expression if it was defined locally. Another big difference is type safety. Before GHC 7.8, the expressions inside a quoted block would not be typechecked. You would still get type errors for the code. Rather than complaining at the definition site it would complain at the usage site however. This makes defining complex functions nearly impossible. But even with GHC 7.8 you get some warnings when splicing code in rather than at it's definition site. For example, it is not checked if a pattern match is exhaustive. Although this check could be done in the type checker, it is done while desugaring to core. I filed this as a bug in GHC (#9113) and it seems that there is work done which would solve this issue.

4.2.2 nextblock

In principle basic blocks are not ordered. The only exception is the first block, which also cannot have any predecessors. In reality however, they are mostly ordered in the order of control flow. With this, it is not strictly necessary to specify which block will be next after a loop as it is mostly the textually next one. This simplifies the syntax somewhat.

```
for <ty1> <var1> in <val1> to <val2>
                 with <ty2> <values> as <var2>
                 { <loop body> }
```

An example of this can be found in figure 3.3, page 23.

LLVM doesn't assume any ordering on basic blocks and it is therefore impossible to implement this functionality directly. Instead, I insert a branch to a nonexisting block at the end of the loop.

```
br label nextblock
```

When processing the list of all basic blocks, this can be replaced with the correct label.

4.2.3 Antiquotation

I have so far ignored how antiquotation is handled. When antiquoting a value, it is usually inserted directly into the AST. This is straight forward for things like `Types`, `Constants` and `Operands`. It would be useful however to also antiquote sequences of code.

The existing code in accelerate-llvm uses the `CodeGen` monad to produce code.

```
data CodeGenState = CodeGenState
  { blockChain          :: Seq Block
  , symbolTable         :: Map Name AST.Global
```

```
  , metadataTable       :: HashMap String (Seq [Maybe Operand])
  , intrinsicTable      :: HashMap String Name
  , next                :: {-# UNPACK #-} !Word
  }
  deriving Show

data Block = Block
  { blockLabel          :: Name
  , instructions        :: Seq (Named Instruction)
  , terminator          :: Maybe (Named Terminator)
  }
  deriving Show

newtype CodeGen a =
  CodeGen { runCodeGen :: State CodeGenState a }
    deriving ( Functor, Applicative, Monad
             , MonadState CodeGenState)
```

This monad collects instructions into unfinished blocks. When all blocks are finished, it is then possible to convert these into a list of `BasicBlocks`. The interesting return values are of type `Operand` or `[Operand]`. To use these in further calculations, they have to be linked to a known value. This could be achieved by a function

```
(.=.) :: Operand -> CodeGen Operand -> CodeGen BasicBlock
```

The first argument has to be a `LocalReference`. There is no way to directly assign a value to a name in LLVM without using an additional instruction. It is however possible to abuse instructions to do exactly this. The **select** instruction can be used in this case. To do this, both branches of the select have to point to the same value. This way the additional instruction can be easily removed via constant propagation.

```
bar :: Int64 -> CodeGen Operand

foo :: Int64 -> Codegen Global
foo y_val =
```

```
let y = LocalReference i64 (Name "y")
in [llg|
  define i64 @foo(i64 %x) {
    $bbsM:(y .=. bar y_val)
    %z = add i64 %x, %y
    ret i64 %z
  }|]
```

4.2.4 Mutable Variables

Although the above methods work, there are some fundamental problems with them. The fact that you have to specify the variables in the loop header and then return the new value is awkward. It abuses the return for something that it is not, namely jumping to the loop header. I could have chosen to use a jump instead, but then there would be no way to update the accumulator. Another more important flaw is the limitation to one value. It is possible, of course, to bundle all needed values together in a struct, but this adds a lot of necessary boilerplate code.

In an ordinary language a for loop would consist of just the loop counter and would handle accumulators through mutable variables. LLVM doesn't support mutable variables however. Since I need however, I implemented support for them through my quasiquoter. The implementation is described in section 4.3.

With this idea, the syntax for the for loop is now reduced to

```
for <ty1> <var1> in <val1> to <val2> { <loop body> }
```

This is a great improvement upon the first approach. The main benefit now is the increased flexibility. It is also more concise and doesn't abuse the return to pass through variables.

As a side effect, the implementation got a lot simpler. It is now no longer necessary to scan for return statements to fill **phi**-instructions for both counter and accumulator.

Instead, the last block in the loop body just jumps to an end-block, where the counter gets increased and then jumps to the head. Now the for-loop can be implemented exactly like shown in figure 4.2.

To allow mutable variables, there has to be a way to initialize them. The syntax for this is

```
<var> = <ty> <val>
```

To implement this, I used the select statement. The above code gets translated into

```
<var> = select i1 true, <ty> <val>, <ty> <val>
```

It always selects the first value. Another way would be to leave the second value undefined.

```
<var> = select i1 true, <ty> <val>, <ty> undef
```

Obviously this adds another unnecessary instruction to the produced code. This is negligible however, as it is easily removed by LLVM's constant propagation.

4.3 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.

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.

```
writeVariable(variable, block, value):
    currentDef[variable][block] <- value

readVariable(variable, block):
    if currentDef[variable] contains block:
        # local value numbering
        return currentDef[variable][block]
    # global value numbering
    return readVariableRecursive(variable, block)
```

Figure 4.3: Implementation of local value numbering

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 Φnodes, this is straight forward, as Figure 4.3 shows.

If the given variable has no definition in the current block, an empty Φ-node is inserted and the local definition is set to the Φ-node. Now the lookup is done recursively and the values are added to the Φ-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 Φ-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 4.4 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

```
readVariableRecursive(variable, block):
    if |block.preds| = 0:
        # First block
        val <- variable
    else:
        # Break potential cycles with operandless phi
        val <- new Phi(block)
        writeVariable(variable, block, val)
    writeVariable(variable, block, val)
    return val
```

Figure 4.4: Implementation of global value numbering

`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.

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,
  blockIncompletePhis
    :: STRef s (M.Map Name (MutableInstruction s)),
  blockPhis :: 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

```
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 4.5: `toSSA`

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 4.5 shows the complete `toSSA` function.

5 Skeletons

The skeletons are one of the core components of an accelerate backend. They provide the functionality of a given array function like **map** or `fold`. They are basically functions that given extra arguments for array environment and higher order function generate a concrete function which can then be compiled. In accelerate-llvm, this process is abstracted through a typeclass.

```
class Skeleton arch where
  generate      :: (Shape sh, Elt e)
                => arch
                -> Gamma aenv
                -> IRFun1 aenv (sh -> e)
                -> CodeGen [Kernel arch aenv (Array sh e)]
  ...
```

Each of the different functions in this class corresponds to a specific skeleton.

When implementing parallel algorithms, it often becomes more complicated. Accelerate was specifically designed to only consist of operations that can be implemented efficiently.

In the following, I will go through the skeletons one by one.

5.1 generate

The simplest skeleton is `generate`. As the name suggests, it produces an array with just a function from index to a value.

```
mkGenerate
    :: forall arch aenv sh e. (Shape sh, Elt e)
    => Gamma aenv
    -> IRFun1 aenv (sh -> e)
    -> CodeGen [Kernel arch aenv (Array sh e)]
```

Before the actual quasiquoted llvm code, we have to set up some values first. The first thing we need are the parameters for the output array and the array environment. The start and stop indices are also required.

```
mkGenerate aenv apply =
  let
      arrOut   = arrayDataOp  (undefined::Array sh e) "out"
      shOut    = arrayShapeOp (undefined::Array sh e) "out"
      paramOut = arrayParam   (undefined::Array sh e) "out"
      paramEnv = envParam aenv
      (start, end, paramGang)
               = gangParam
```

The last piece before the actual code are some declarations for used types and local variables.

```
      intType  = typeOf (int :: IntegralType Int)

      r        = locals (undefined::e) "r"
      i        = local intType "i"
      ix       = locals (undefined::sh) "ix"
```

The rest is then relatively self-explaining.

```
in
makeKernelQ "generate" [llgM|
  define void @generate
  (
      $params:paramGang,
      $params:paramOut,
      $params:paramEnv
  )
  {
      for $type:intType %i in $opr:start to $opr:end
      {
          ;; convert to multidimensional index
          $bbsM:(ix .=. indexOfInt shOut i)
          ;; apply generator function
          $bbsM:(r  .=. apply ix)
          ;; store result
          $bbsM:(writeArray arrOut i r)
      }
      ret void
  }
|]
```

vectorization

If `generate` vectorizes is very dependent on the supplied function. A function which randomly references a different manifest array would not vectorize for example. The challinging part for the optimizer are the multidimensional indices. This means the chance of vectorization is much higher if the target is just a vector.

5.2 map

map works similar to `generate`. In fact it can be easily implemented using `generate`. One important difference between `generate` and **map** is how they access the array. `generate` works with multidimensional indices. This is not necessary with **map** however, as **map** keeps indices identical.

```
mkMap :: forall t aenv sh a b. (Elt b, Elt a)
      => Gamma aenv
      -> IRFun1    aenv (a -> b)
      -> IRDelayed aenv (Array sh a)
      -> CodeGen [Kernel t aenv (Array sh b)]
```

The following is the core part of the **map** skeleton. I omit the rest of the function as the difference to `generate` is mostly mechanical.

```
makeKernelQ "map" [llgM|
    define void @map
    (
        $params:paramGang,
        $params:paramOut,
        $params:paramEnv
    )
    {
        for $type:intType %i in $opr:start to $opr:end
        {
            $bbsM:(x .=. delayedLinearIndex [i])
            $bbsM:(y .=. apply x)
            $bbsM:(writeArray arrOut i y)
        }
        ret void
    }
  |]
```

vectorization

map vectorizes far better than `generate`. If the supplied array is manifest, no index conversion is necessary. If the function doesn't contain branches, then it is very likely that vectorization is possible.

5.3 transform

`transform` is closely related to map. The difference is, that the index is not preserved. Instead there is another function which projects an index in the target array into an index in the source array.

```
defaultTransform
    :: (Shape sh, Shape sh', Elt a, Elt b, Skeleton arch)
    => arch
    -> Gamma aenv
    -> IRFun1    aenv (sh' -> sh)
    -> IRFun1    aenv (a -> b)
    -> IRDelayed aenv (Array sh a)
    -> CodeGen [Kernel arch aenv (Array sh' b)]
```

In fact **map** can be easily implemented using transform:

```
defaultMap arch aenv f a = transform arch aenv return f a
```

We keep a seperate implementation of **map** however for performance reasons. The specific reason for this is that **map** preserves the the indices. Since `transform` doesn't have this property, we simply implement it in terms of `generate`.

```
defaultTransform arch aenv p f IRDelayed{..} =
  generate arch aenv $ \ix -> do
    ix' <- p ix
    a   <- delayedIndex ix'
    f a
```

vectorization

If `transform` vectorizes is very dependent on the functions involved. In case of a simple identity (**map**) it will vectorize just fine. It is however really not at all guaranteed, that it will vectorize. A simple counter example is the following.

```
let a0 = use (Array (Z :. 10) [1,2,3,4,5,6,7,8,9,10])
in backpermute
     (Z :. 10)
     (\x0 -> Z :. mod (1+(indexHead x0), indexHead (shape a0)))
     a0
```

All this code does is rotate the array to the right by one element and then adding one. This code, if optimized by hand, could be easily vectorized.

To give another positive example, let's look at the reversal of an array.

```
let a0 = use (Array (Z :. 10) [1,2,3,4,5,6,7,8,9,10])
in transform
     (shape a0)
     (\x0 -> Z :. -1 + ((indexHead (shape a0)) - (indexHead x0)))
     (\x0 -> 1 + x0)
     a0
```

This vectorizes just fine. LLVM uses **shufflevector** to achieve this.

```
...
%reverse = shufflevector <4 x i64> %wide.load,
                         <4 x i64> undef,
                         <4 x i32> <i32 3, i32 2, i32 1, i32 0>
...
```

5.3.1 backpermute

Accelerate handles a special case of `transform` as `backpermute`. This is essentially just a transform without the map functionality. We don't have a specific implementation of `backpermute`, but rather use `transform`.

```
defaultBackpermute
    :: (Shape sh, Shape sh', Elt e, Skeleton arch)
    => arch
    -> Gamma aenv
    -> IRFun1    aenv (sh' -> sh)
    -> IRDelayed aenv (Array sh e)
    -> CodeGen [Kernel arch aenv (Array sh' e)]
defaultBackpermute arch aenv p a
    = transform arch aenv p return a
```

5.4 permute

`permute` is analog to `backpermute`. Instead of projecting every cell of the target array onto a cell in the source array, the opposite is done. Every cell in the source array is projected onto one or no cell in the target array. There is no assumption being made about this projection. This means there can be either multiple or no cell pointing to a single output cell. To deal with no projection the output array is initialized with a given array. To deal with multiple cells, there is a function to combine the existing value with the new one.

```
mkPermute
    :: forall arch aenv sh sh' e. (Shape sh, Shape sh', Elt e)
    => Gamma aenv
    -> IRFun2 aenv (e -> e -> e)
    -> IRFun1 aenv (sh -> sh')
    -> IRDelayed aenv (Array sh e)
    -> CodeGen [Kernel arch aenv (Array sh' e)]
```

To get the values, the loop goes over all the elements in the source array. This is then projected into the target array.

```
makeKernelQ "permute" [llgM||
  define void @permute
  (
      $params:paramGang,
      $params:paramBarrier,
      $params:paramOut,
      $params:paramEnv
  )
  {
      for $type:intType %i in $opr:start to $opr:end
      {
          $bbsM:(ix .=. indexOfInt shOut i)
          $bbsM:(ix' .=. permute ix)
...
          }
      }
      ret void
  }|||]
```

There is however an option that the index is not projected at all. This is signalled by a magic value `ignore`. In this case the inner loop is empty.

```
$bbsM:(c1 .=. all (uncurry (neq int)) (zip ix' ignore))
if %c1 {
...
}
```

If the value is not ignored, the permutation step will be executed.

```
$bbsM:(old .=. readArray arrOut dst)
$bbsM:(new .=. delayedLinearIndex [i])
$bbsM:(val .=. combine new old)
$bbsM:(writeArray arrOut dst val)
```

This would be enough in a single-threaded context. In a multi-threaded context this is not enough however. In that case, this central step is a critical section. This means it has to be executed atomically. To achieve this, we have a vector of essentially booleans. This vector has an element for every element of the result array. We use special atomic memory operation to use these for a spinlock.

First we have to determine the specific memory cell to use.

```
%baddr = getelementptr i8* $opr:barrier,
                       $type:intType $opr:dst
```

With this, we can now look at the the lock is taken.

```
%c2 = i1 true
while %c2 {
    %lock = atomicrmw volatile xchg i8* %baddr, i8 1 acq_rel
    %c2 = icmp eq i8 %lock, 0
}
```

atomicrmw `volatile` **xchg** excanges the current value with a new value and returns the old one. This way we know if the lock was taken before or not. `volatile` signals that the operations can't be reordered. This pattern is common enough, that modern CPUs have optimizations in place. The associated cache line will be shared between all CPUs/cores. This was there is no unnecessary bus traffic.

The lock release is less involved.

```
store atomic volatile i8 0, i8* %baddr release, align 1
```

It is possible to use a regular store here. This will be translated into just a MOV instruction. This is possible due to subtle memory ordering rules which allow this, even though MOV is not a full memory barrier. In case of other architectures like ARM a MOV is not sufficient here.

vectorization

Vectorization is not possible in case of `permute`. In every step, the same cell in the output array is both read and written. The only way this could possibly vectorize is if this index was predictable.

That said, the current implementation cannot possibly vectorize. What blocks the vectorization is the synchronization via spinlocks. It may be beneficial to supply an additional implementation for single-threaded execution. This would not need the extra array of semaphores. I leave this evaluation for future work.

5.5 fold

In Haskell, a fold reduces a list to a single value. The equivalent would be to reduce a vector to a scalar. While Accelerate also supports this, it has a more general view. It is possible to also reduce an Array of arbitrary dimensionality. The result will then be an array with it's inner dimension reduced.

```
mkFold
    :: forall t aenv sh e. (Shape sh, Elt e)
    => Gamma aenv
    -> IRFun2    aenv (e -> e -> e)
    -> IRExp     aenv e
    -> IRDelayed aenv (Array (sh:.Int) e)
    -> CodeGen [Kernel t aenv (Array sh e)]
```

The natural unit of computation for a fold is one element in the output array. This degree of parallelism can be achieved without multiple passes.

```
makeKernelQ "fold" [llgM|
  define void @fold
  (
      $params:paramGang,
```

```
      $params:paramStride,
      $params:paramOut,
      $params:paramEnv
  )
  {
    %sz = mul $type:intType $opr:start, $opr:n

    for $type:intType %sh in $opr:start to $opr:end
    {
        %next = add $type:intType %sz, $opr:n
        $bbsM:(x .=. seed)

      reduce:
        for $type:intType %i in %sz to %next
        {
            $bbsM:(y .=. delayedLinearIndex [i])
            $bbsM:(x .=. combine x y)
        }

        $bbsM:(writeArray arrOut sh x)
        %sz = $type:intType %next
    }
    ret void
  }
|]
```

This works great if the output still has a dimension left. If the output is a skalar however, it doesn't work that well. Instead, the array is reduced in multiple steps. The first step is to create an array of partial sums. This array can then be reduced by a single thread sequentially.

5.5.1 Segmented fold

The segmented fold is a generalization of the regular fold. The regular fold reduces the inner dimension to a single value. The segmented fold reduces segments of the inner dimension into single values. This way the dimensionality of the resulting array is the same as the incoming array. The degree of parallelism is increased by the same amount as there are segments.

vectorization

In general, `fold` doesn't vectorize very well. `fold` requires it's input function to be associative. For LLVMs loop vectorization to work however, it also needs to be commutative. Many associative functions are also commutative however. This means that these cases will vectorize well.

5.6 scan

Scan (or prefix sum) is closely related to fold. Instead of just providing one value at the end however, it produces all the values in between. In contrast to fold, scan only operates on vectors. This makes implementation somewhat easier, as there is no special case for this as with `foldAll`. Scan also comes in different flavours: **`scanl`**, **`scanr`**, **`scanl'`** and **`scanr'`**. I will focus only on **`scanl`** here, as the differences are mostly mechanical.

```
mkScanl
    :: forall t aenv e. (Elt e)
    => Gamma aenv
    -> IRFun2    aenv (e -> e -> e)
    -> IRExp     aenv e
    -> IRDelayed aenv (Vector e)
    -> CodeGen [Kernel t aenv (Vector e)]
```

As with `fold`, the implementation of prefix sums differs between non-parallel and parallel. Scan is difficult to implement as a parallel algorithm. Every element depends on its direct predecessor. The approach I took was:

1. divide the array into chunks
2. build the sum of each chunk
3. build the prefix sums over those sums
4. use the prefix sums as starting point to build the final array

This approach requires 2 read operation and 1 write operation asymptotically. I decided to build combine the last 2 steps into one. The last step is done completely in parallel, so this leads to some extra work. Depending on the size of the array this may be faster as calling into a kernel has some overhead. I leave this evaluation for future work.

The first 2 steps of the process are basically identical to the multidimensional fold. The rest forms one big kernels, as I essentially combine 2 separate operations into one.

First look at the function parameters.

```
makeKernelQ "scanl" [llgM|
  define void @scanl
  (
      $params:paramGang,
      $type:intType %lastChunk,
      $type:intType %chunkSize,
      $type:intType %sz,
      $params:paramTmp,
      $params:paramOut,
      $params:paramEnv
  )
  {
      for $type:intType %i in $opr:start to $opr:end
      {
...
      }
```

```
      ret void
  }
|]
```

The gang parameters (start, stop) in this case are measured not in single elements, but in chunks. The size of these chunks is given by `%chunksize`. The total length and the number of chunks is also provided. This is necessary to handle the last chunk as it is not of full length.

The first step then is the scan over the previously computed sums. The seed element is also written to the first array index in this step.

```
;; sum up the partial sums to get the first element
$bbsM:(acc .=. seed)

for $type:intType %k in 0 to %i
{
     $bbsM:(x .=. delayedLinearIndex tmpD [k])
     $bbsM:(acc .=. combine x acc)
}

;; check if this is the first chunk
%c2 = icmp eq $type:intType %i, 0
;; if it is, write the seed to memory
if %c2 {
    %c3 = icmp ne $type:intType %sz, 0
    if %c3 {
        $bbsM:(writeArray arrOut zero acc)
    }
}
```

The outer loop just gives the chunk number. This has to be converted into a usable range first.

```
;; calculate the start of the current chunk
```

```
%ix    = mul $type:intType %i,  %chunkSize
;; Determine the end of the current chunk.
;; If this is the last chunk,
;; then this is the same as the end of the array.
%last1 = add $type:intType %ix, %chunkSize
%c1    = icmp eq $type:intType %i, %lastChunk
%last  = select i1 %c1, $type:intType %sz, $type:intType %last1
```

With the correct range and the prefixsum of all previous elements, the rest is easy.

```
for $type:intType %k in %ix to %last
{
    %k1 = add $type:intType %k, 1
    $bbsM:(x .=. delayedLinearIndex inD [k])
    $bbsM:(acc .=. combine acc x)
    $bbsM:(writeArray arrOut k1 acc)
}
```

I didn't define a seperate function for the sequential case. It is easy though to use the exact same function as above though. It is only necessary to supply an empty array of precalculated sums. This way the scan over that array yields the just seed element. This is then used to compute the prefix sum directly.

vectorization

The steps equivalent to a fold vectorize very well. The this is in essence all but the last step. The actual prefix sum cannot be vectorized, as all the intermediate values have to be written to memory.

5.7 stencil

Stencil operations are basically a generalization of map. The difference is that the focus is not on a single element. Instead it also looks at the surrounding cells.

```
mkStencil
  :: forall arch aenv sh stencil a b.
     (Elt b, Stencil sh a stencil)
  => Gamma aenv
  -> Proxy (stencil,a)
  -> IRFun1 aenv (stencil -> b)
  -> Boundary (IRExp aenv a)
  -> CodeGen [Kernel arch aenv (Array sh b)]
```

This is leads to a problem however. While **map** is always guaranteed to read in bounds, this can't be said about stencils. There are multiple ways to avoid this problem. Accelerate tries to capture the different options via the `Boundary` type.

```
data Boundary a
  = Clamp
  | Mirror
  | Wrap
  | Constant a
```

They can be divided into 2 groups. All except `Constant` rely on index manipulation. `Constant` on the other hand does no read at all if the read would be out of bounds. I use a single function `access` to deal with all of these cases. The returned value is a the value read at the given index modulo the boundary condition. I start by applying the offset to the index. If the offset is 0 in every direction however, the read can be done directly.

```
access bndy sh ix arr off =
  if all (==0) off
    then do
      i <- intOfIndex sh ix
```

```
    readArray arr i
  else do
    ix' <- zipWithM (add int . (constOp . num int)) off ix
```

This gives me the correct cell to do the read on. Then I group the Boundary into `Constant` and the rest. In this step I also replace the boundary with its corresponding handler function.

```
let op = case bndy of
      Constant as -> Left as
      Clamp       -> Right clamp
      Mirror      -> Right mirror
      Wrap        -> Right wrap
```

Now all is left is to do the boundary checks and the read. In case of `Clamp`, `Mirror` and `Wrap` this is straight forward.

```
case op of
  Right op' -> do
    ix'' <- op' sh ix'
    i <- intOfIndex sh ix''
    readArray arr i
```

In case of `Constant` however this is a little more complicated.

```
  Left as -> do
    as'       <- as
    ix''      <- wrap sh ix'
    i         <- intOfIndex sh ix''
    c         <- inRange sh ix'
    zipWithM (\a x -> instr (typeOfOperand a)
             (Select c x a [])) as' xs
```

I prepare both the constant and the read value. I then do the bounds check and select either value depending on the result. To avoid out of bounds reads, I wrap the index

before I read it. This is necessary since it is fairly easy to construct stencil functions that would end in a segfault otherwise.

The complete stencil is then constructed via the function `stencilAccess`.

```
stencilAccess
  :: forall a b sh aenv stencil. (Shape sh, Stencil sh a stencil)
  => Proxy (stencil,a)
  -> Proxy sh
  -> [Operand]
  -> [Operand]
  -> Boundary (IRExp aenv a)
  -> IRFun1 aenv (sh -> stencil)
stencilAccess _ _ sh arr bndy ix = do
  let off  = offsets (undefined :: Fun aenv (stencil -> a))
                     (undefined :: OpenAcc aenv (Array sh a))
      op   = map (constOp . num int) . reverse . shapeToList
      off' = map op off
  stencil <- mapM (access bndy sh ix arr) off'
  return (concat stencil)
```

The `Proxy` values immediately look out of place. It seems that the types are already present in the `Boundary` and the result type. They are necessary however, as those are just type synonyms. Apart from this oddity, there isn't much happening in this function. It mainly combines existing offsets and applies the `access` function.

All that remains now is the actual skeleton code. This looks very similar to the **map** skeleton.

```
makeKernelQ "stencil" [llgM|
  define void @stencil
  (
    $params:paramGang,
    $params:paramIn,
    $params:paramOut,
```

```
    $params:paramEnv
  )
  {
    for $type:intType %i in $opr:start to $opr:end
    {
      $bbsM:(ix .=. indexOfInt shOut i)
      $bbsM:(x .=. (stencilAccess stencilT shT shIn arrIn bndy ix
                    >>= apply))
      $bbsM:(writeArray arrOut i x)
    }
    ret void
  }
|]
```

Unfortunately I wasn't able to store the result of `stencilAccess`, as I don't have `(Elt stencil)` in context. This is mainly due to the use of type families in the definition of the `Stencil` typeclass. The code for `stencil2` is analogous.

vectorization

The stencil code doesn't vectorize at all at the moment. The reason for this is the boundary. If the whole stencil is guaranteed to be in bounds, the bounds check could be omitted however. This way the read would be guaranteed to vectorize.

6 Performance

In the previous chapter I explained the implementation of the skeletons. In this chapter I will analyze the performance of Accelerate as a whole while using those skeletons.

The Benchmarks were run on a dual CPU Xeon E5-2650 (16 cores, 2GHz, 64GB RAM) running GNU/Linux (Ubuntu 12.04 LTS). Programs are compiled with ghc-7.8.3 and llvm-3.4.2. All programs are executed on all 16 cores unless stated otherwise.

6.1 Ray Tracer

Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects. The technique is capable of producing a very high degree of realism compared to typical scanline rendering methods, as ray tracing algorithms are able to simulate optical effects such as reflection and refraction, scattering, and dispersion phenomena. This increased accuracy has a larger computational cost, which is why ray tracing is not typically used for real time rendering applications such as computer games. Figure 6.1 shows the result of rendering a scene using a ray-tracer implemented in Accelerate. The algorithm computes the colour of a pixel by casting a ray from that point in a specified direction, and tracing its interaction with the objects in the scene.

After fusion, this ends up being just a single generate kernel. Figure 6.2a shows the runtime of the ray-tracer for different image sizes. The comparison here is against a Repa implementation of the same algorithm. Independent of the image size, the Accelerate version is about 45% faster than the Repa version.

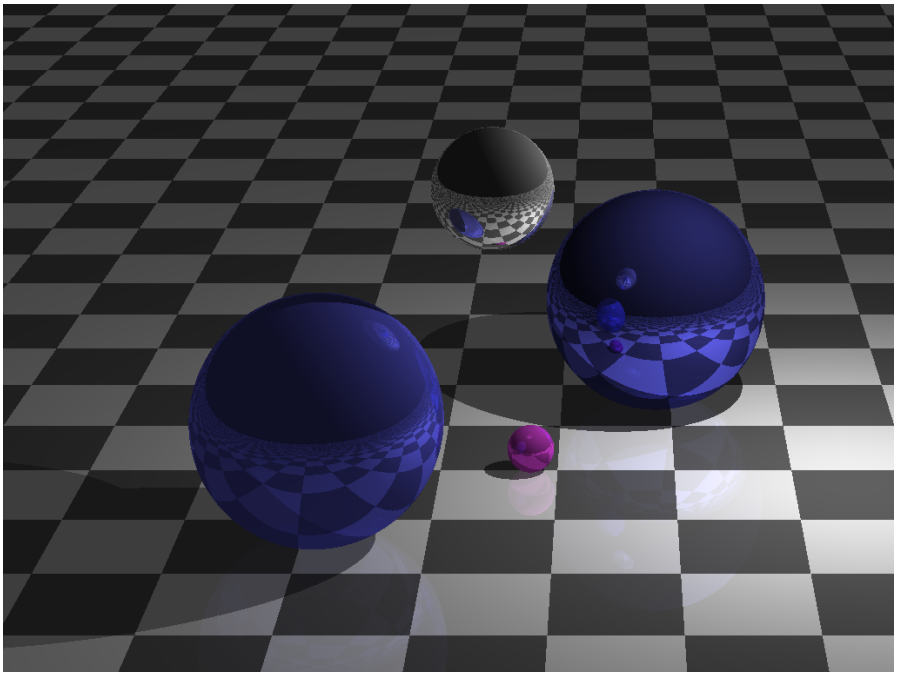
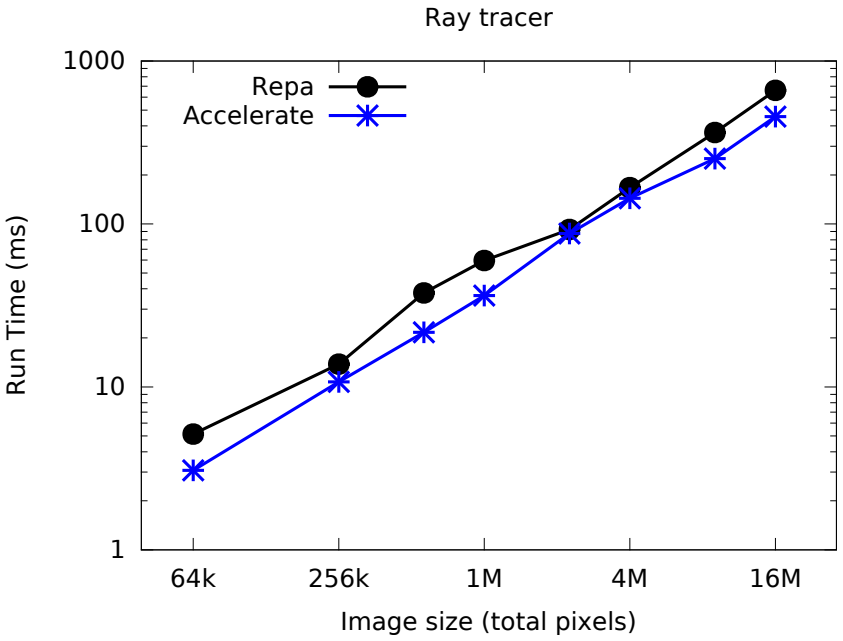
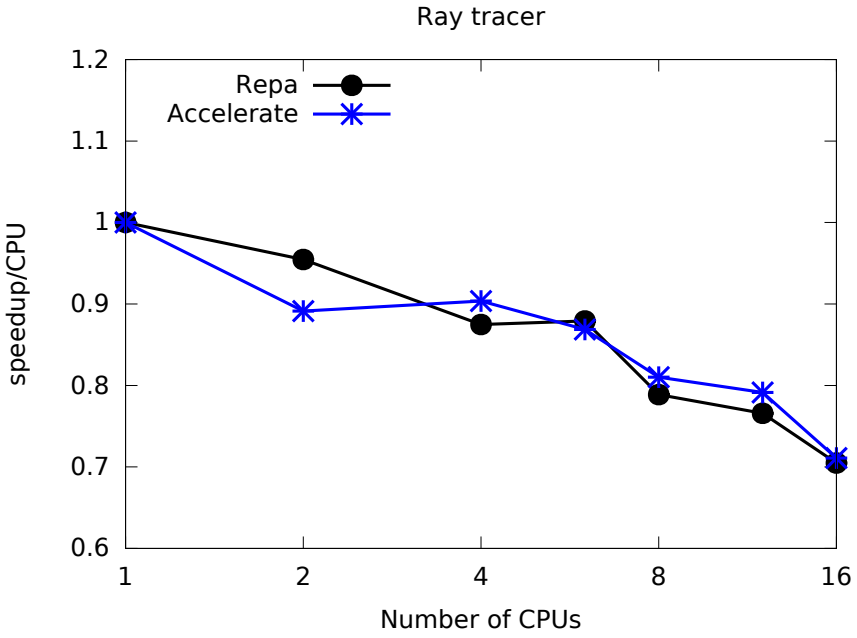


Figure 6.1: Output of raytracer

Accellerate-llvm has very heavy infrastructure in place to synchronize threads. The intuition is, that this might lead to decreased scalability. Figure 6.2b shows the speedup per core using the raytracer. As you can see, Accelerate performs basically identical to Repa in this regard.



(a) Runtimes



(b) Speedup per CPU

Figure 6.2: Benchmarks raytracer

6.2 N-Body

The n-body example simulates the Newtonian gravitational forces on a set of massive bodies in 3D space, using the naïve $\mathcal{O}\left(n^2\right)$ algorithm. In a data-parallel setting, the natural way to express this algorithm is first to compute the forces between every pair of bodies, before adding the forces applied to each body using a segmented sum. Figure 6.3 shows the results of this program in different settings.

After fusion, this example consists of a `generate` and a **`map`** kernel. Both of these vectorize well. This is clear when looking at the difference in runtime between the vectorized and unvectorized kernels executed in a single thread. The difference between them is almost a factor of 8, which is the vector width for floats on the benchmark machine.

Another interesting point here is how it performs against an implementation using `Data.Vector`. For small sizes, `Data.Vector` is faster than Accelerate. This is due to the overhead Accelerate brings to the table. For bigger sizes Accelerate is significantly faster however.

When running with `+RTS -N16`, it doesn't scale quite as well as the raytracer for smaller sizes. This is likely due to memory transfers. The Benchmark machine is a dual CPU server and therefore can't share the on-chip cache between all available cores. In fact the runtime is identical whether executed with 8 or 16 cores. On very large datasets (>12000 bodies) this is no longer the case and the extra parallelism is noticeable.

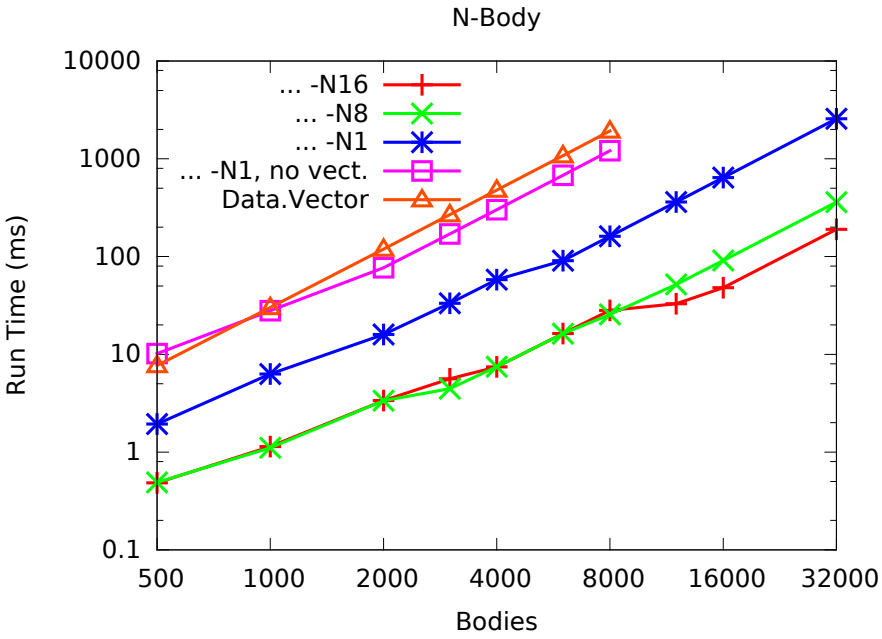


Figure 6.3: Benchmark N-Body

6.3 Canny Edge Detection

The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986. Canny edge detection is a four step process.

1. A Gaussian blur is applied to clear any speckles and free the image of noise.
2. A gradient operator is applied for obtaining the gradients' intensity and direction.
3. Non-maximum suppression determines if the pixel is a better candidate for an edge than its neighbours.
4. Hysteresis thresholding finds where edges begin and end.

An example of the algorithm in action is shown in figure 6.4.

The Canny Edge Detection consists mainly of stencil convolutions. These are very specialized operations. Figure 6.5 shows the run time for different image sizes. The Repa

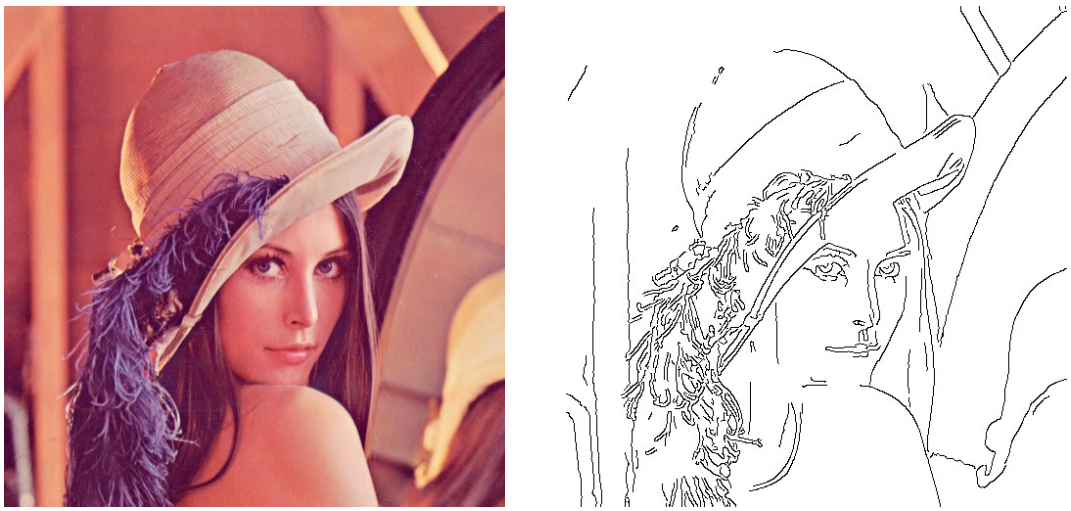


Figure 6.4: Canny Edge Detection

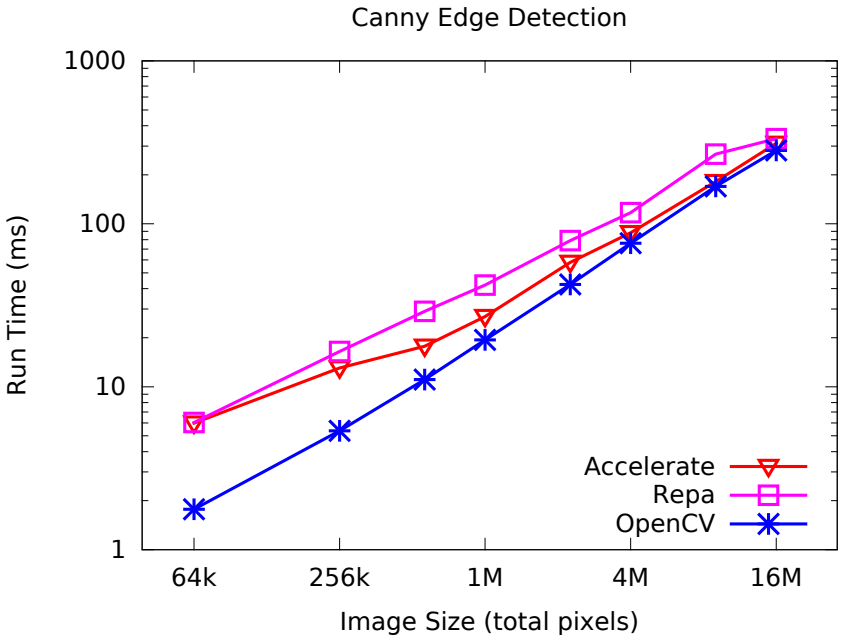


Figure 6.5: Benchmark Canny Edge Detection

and Accelerate versions both execute on 16 cores while the OpenCV version only uses one CPU. Accelerate is consistently faster than Repa in this benchmark. On small image sizes, the OpenCV version, even though only running on 1 core, is significantly faster than both Accelerate and Repa. With larger image sizes, this difference becomes smaller. One important difference is, that OpenCV is carefully optimized to vectorize well. The skeleton code for stencils is barely optimized at all and hence doesn't vectorize. Repa cannot vectorize for technical reasons.[LK12]

6.4 Dot product

Dot product, or scalar product, is an algebraic operation that takes two equal-length sequences of numbers and returns a single number by computing the sum of the products of the corresponding entries in the two sequences.

$$\begin{pmatrix} x_0 \\ ... \\ x_{n-1} \end{pmatrix} \times \begin{pmatrix} y_0 \\ ... \\ y_{n-1} \end{pmatrix} = \sum_{i=0}^{n-1} x_i * y_i$$

Figure 6.6 shows the runtime of the dot product. Before Accelerate can compute it's result, the kernels have to be compiled first. This takes a considerable amount of time – longer than the whole computation using `Data.Vector`. The time needed for this step is however constant independent of the array size and can be mostly avoided when using `run1`.

Only looking at the runtime of the computation, Accelerate is faster than the sequential `Data.Vector` implementation. It is however significantly slower than Repa. Accelerate has a much higher runtime overhead compared to Repa, which explains some of the gap. When distributing the work to the different workers, it is split into smaller chunks. As this benchmark has a particularly tight inner loop, these chunks are likely smaller than optimal.

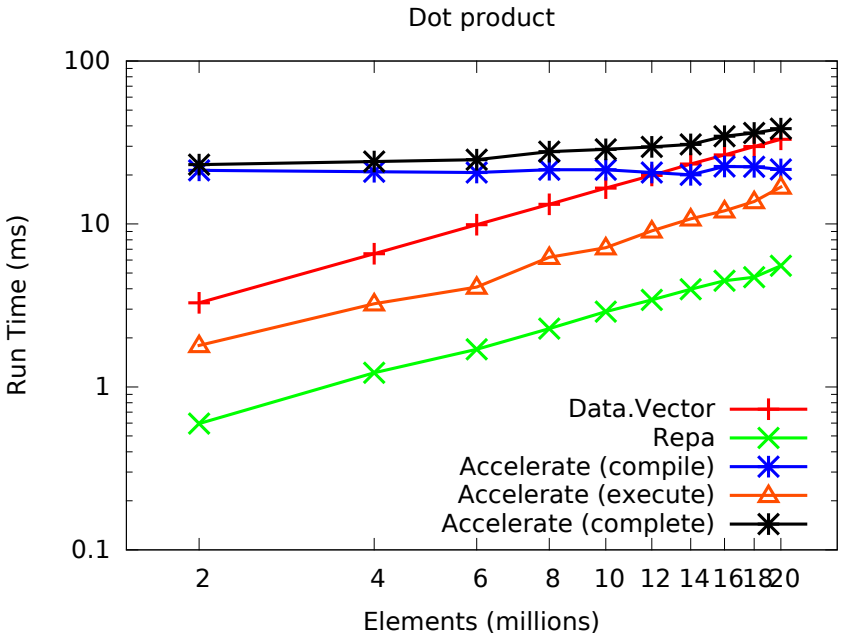


Figure 6.6: Benchmark Dot Product

6.5 Online Compilation

As seen in the dot product benchmark, Accelerate has a high overhead in the compilation step. To understand what exactly the issue is, let's look a bit closer.

The function responsible for this compilation is `compileForNativeTarget` shown in figure 6.8.

Figure 6.7 shows the runtime of each step. All runtimes are averaged over 10 runs.

Looking at the times, there are 3 main cost centres.

- withModuleFromAST
- optimizeModule
- getGlobalFunctions

Step	Time (ms)
getTarget	0.098
llvmOfAcc	0.0771
startFunction	0.0403
withContext	0.0879
withModuleFromAST	6.8032
withMachine	0.2997
libinfo	0.0801
optimizeModule	4.0379
withMCJIT	0.05
withModuleInEngine	0.2374
getGlobalFunctions	9.8508

Figure 6.7: Detailed runtime of online compilation

These all live in the LLVM API and are therefore mostly independent of the Accelerate implemenation. It may be possible to reduce these through options to LLVM, but this is beyond the scope of this thesis. Another interesting aspect is, that the construction of the AST doesn't take up any significant amount of time.

```
compileForNativeTarget
  :: DelayedOpenAcc aenv a -> Gamma aenv
  -> LLVM Native (ExecutableR Native)
compileForNativeTarget acc aenv = do
  target <- gets llvmTarget

  -- Generate code for this Acc operation
  let Module ast = llvmOfAcc target acc aenv
      triple     = fromMaybe "" (moduleTargetTriple ast)
      datalayout = moduleDataLayout ast

  -- Lower the generated LLVM AST all the way
  -- to JIT compiled functions.
  fun <- liftIO . startFunction $ do
    ctx     <- ContT withContext
    mdl     <- runErrorCont $ withModuleFromAST ctx ast
    machine <- runErrorCont withNativeTargetMachine
    libinfo <- ContT $ withTargetLibraryInfo triple

    liftIO $ optimiseModule datalayout (Just machine)
                            (Just libinfo) mdl

    mcjit <- ContT $ withMCJIT ctx opt model ptrelim fast
    exe   <- ContT $ withModuleInEngine mcjit mdl
    liftIO $ getGlobalFunctions ast exe

  return $! NativeR fun
```

Figure 6.8: Implementation of `compileForNativeTarget`

7 Conclusion

Prior to this thesis, only a handful of skeletons were implemented in the llvm backend to accelerate. These worked, but were hard to read as the tool used relied on manipulating blocks directly. In this thesis I presented an alternative approach to code generation for llvm. I implemented the remaining skeletons and rewrote the existing ones using my quasiquoter. The resulting code is much more legible and therefore maintainable than the purely monadic approach taken by previous work.

One reason for using LLVM as the target is, that it is very clear what the produced code will look like on assembler-level. I was important for me not to compromise this advantage. I parse LLVM Instructions directly without modification, so it is guaranteed that the user has absolute control over the generated code. To improve legibility and maintainability, I introduced concepts from high-level languages like mutable variables and control structures. These are purely an addition to the existing LLVM language and are translated into LLVM code by the quasiquoter.

Using a foreign library in Haskell is often tedious, as the main documentation doesn't quite line up with the Haskell bindings. LLVM is no exception to this, although the AST used is very well documented. This where the quasiquoter shines, as the instructions are exactly the same as documented in the language reference.

Most languages model variables as cells in memory. This is fine, as LLVM is capable to lower these into register values if possible. The code already present in the LLVM backend didn't use this method however, so the quasiquoter doesn't use this mechanism. Instead, it treats regular LLVM values as variables that can be re-assigned at any time. To this end, I implemented a transformation back into legal LLVM code. The resulting language has everything you would expect from an imperative language like mutable

variables and control structures. In this way, it "feels" much like programming in C, while still having a clear representation in LLVM IR.

7.1 Related Work

Haskell offers a range of ways to implement operations on arrays effectively. The one most closely resembling Accelerate is Repa. As in the LLVM backend described in this thesis, it uses gang workers to execute computations on multiple CPUs concurrently. The important difference between Accelerate and Repa is the execution model. While Repa computations behave like any other Haskell computation, Accelerate behaves more like running a Monad. Since Accelerate is device-independent, it has to compile the needed functions at runtime before the actual computation can occur.

In terms of code generation, C– most closely resembles the language I created. It was designed as an intermediate language for compilers and is in use in GHC. It's design is inspired by C, but it replaces many language elements for clearer definitions. Like LLVM, it's type system is deliberately designed to reflect constraints of the underlying hardware. There is however no clear standard implementation for C–, making it more difficult to use than LLVM.

7.2 Future Work

I developed the quasiquoter mainly as a tool to use for skeleton code generation. This means, that the main focus was on generating code for complete functions. It should be possible to use it to generate elements of code that could then be spliced in, but I haven't explored this.

The SSA transformation is not optimal. It produces code with far too many **`phi`** instructions. LLVM is capable of eliminating those redundant instructions, but an optimization pass is necessary for this. If the code is to be printed directly however, this produces non-optimal code.

With my contribution, the LLVM backend is almost functionally complete and passes all existing tests. The performance is promising, outrunning Repa in most cases. There is however much room for improvement as not all skeletons vectorize as well as they possibly could. Stencil convolutions for example don't vectorize at all at the moment, although they could if implemented with this in mind.

Although it is functionally almost complete, it is not ready for a release yet. The major part still missing is the support for FFI (#183). There are also still a number of bugs that have to be addressed first. These range from space leaks (#181) to random segfaults (#179). A list of open issues can be found at[1].

[1] `https://github.com/AccelerateHS/accelerate/labels/llvm%20backend`

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%5C_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

[Lat02] Lattner, 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

[LK12] Lippmeier, Ben ; Keller, Gabriele: Efficient parallel stencil convolution in Haskell. In: *ACM SIGPLAN Notices* 46 (2012) Nr. 12, pp. 59–70. URL: http://research.microsoft.com/en-us/um/people/simonpj/papers/ndp/Stencil.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