

# *Master Thesis*

## An LLVM Backend for Accelerate



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

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

Kiel, July 1, 2014



# Selbstständigkeitserklärung

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

July 1, 2014

---

Timo von Holtz



# Todo list

Write Introduction . . . . .	1
description of types . . . . .	4
longer introduction to monadic code generation . . . . .	7
write about llvm-general-typed . . . . .	7
update figure to correct ordering of bbs . . . . .	9



# Contents

<b>1. Introduction</b>	<b>1</b>
<b>2. Technologies</b>	<b>3</b>
2.1. LLVM . . . . .	3
2.1.1. LLVM IR . . . . .	3
2.1.2. Vectorization . . . . .	5
2.2. Accelerate . . . . .	5
<b>3. Contributions</b>	<b>7</b>
3.1. llvm-general . . . . .	7
3.2. llvm-general-quote . . . . .	7
<b>4. Implementation</b>	<b>13</b>
4.1. Quasiquoter . . . . .	13
4.2. Extension for-loop . . . . .	13
4.3. SSA . . . . .	13
<b>5. Skeletons</b>	<b>15</b>
5.1. map . . . . .	15
5.2. fold . . . . .	15
5.3. scan . . . . .	15
<b>6. Conclusion</b>	<b>17</b>
6.1. Related Work . . . . .	17
<b>A. Listings</b>	<b>19</b>
<b>References</b>	<b>23</b>





# List of Figures

2.1. sum as a C function . . . . .	3
2.2. sum as a LLVM . . . . .	4
3.1. Monadic generation of for loop . . . . .	8
3.2. For Loop using <i>llvm-general-quote</i> . . . . .	9
3.3. Expanded For Loop . . . . .	10
3.4. For Loop using <i>llvm-general-quote</i> and SSA . . . . .	10
3.5. Expanded For Loop (SSA) . . . . .	11



## List of Tables



# 1. Introduction

Write Introduction



## 2. Technologies

### 2.1. LLVM

LLVM[10] 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.[1, 14] A program is said to be in SSA form if each of its variables is defined exactly once, and each use of a variable is dominated by that variable's definition. SSA form greatly simplifies many dataflow optimizations because only a single definition can reach a particular use of a value, and finding that definition is trivial.

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

The first obvious difference is how the for-loop is translated. In LLVM every function is divided into basic blocks. A basic block is a continuous stream of instructions (add, mult, call, ...) with a terminator at the end. Terminators can either be used to jump to another block (branch) or return to the calling function.

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

Figure 2.1.: sum as a C function

```
1 define double @sum(double* %a, i32 %length) {  
2 entry:  
3   %cmp4 = icmp sgt i32 %length, 0  
4   br i1 %cmp4, label %for.body, label %for.end  
5  
6 for.body:                                ; preds = %entry, %for.body  
7   %indvars.iv = phi i64 [ %indvars.iv.next, %for.body ], [ 0, %entry ]  
8   %x.05 = phi double [ %add, %for.body ], [ 0.000000e+00, %entry ]  
9   %arrayidx = getelementptr inbounds double* %a, i64 %indvars.iv  
10  %0 = load double* %arrayidx, align 8  
11  %add = fadd double %x.05, %0  
12  %indvars.iv.next = add nuw nsw i64 %indvars.iv, 1  
13  %lftr.wideiv = trunc i64 %indvars.iv.next to i32  
14  %exitcond = icmp eq i32 %lftr.wideiv, %length  
15  br i1 %exitcond, label %for.end, label %for.body  
16  
17 for.end:                                ; preds = %for.body, %entry  
18  %x.0.lcssa = phi double [ 0.000000e+00, %entry ], [ %add, %for.body ]  
19  ret double %x.0.lcssa  
20 }
```

Figure 2.2.: sum as a LLVM

The  $\Phi$ -nodes in the SSA are represented with **phi**-instructions. These have to precede every other instruction in a basic block.

## Types

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

description of  
types

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.

#### Fast-Math Flags

To vectorize code the operations involved need to be associative. When working with floating point numbers, this property is violated. To vectorize the code regardless, LLVM has the notion of fast-math-flags. These tell the optimizer to assume certain properties that aren't true in general.

Available flags are[11]:

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

## 2.2. Accelerate

Similar to Repa[8], uses gang workers.[3]



## 3. Contributions

### 3.1. llvm-general

- Targetmachine (Optimization)
- fast-math

### 3.2. llvm-general-quote

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

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.

longer introduction to monadic code generation

Another approach would be to use an EDSL.

I propose a third approach using quasiquotation[12]. The idea behind quasiquotation is, that you can define a DSL with arbitrary syntax, which you can then directly transform into Haskell data structures. This is done at compile-time, so you get the same type safety as writing the AST by hand.

write about llvm-general-typed

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 as 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 |]`

```

for :: Type                                -- type of the index
      → Operand                             -- starting index
      → (Operand → CodeGen Operand)         -- loop test to keep going
      → (Operand → CodeGen Operand)         -- increment the index
      → (Operand → CodeGen ())              -- body of the loop
      → CodeGen ()

for ti start test incr body = do
  loop ← newBlock "for.top"
  exit ← newBlock "for.exit"

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

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

  body i
  i' ← incr i
  c' ← test i'
  bot ← cbr c' loop exit
  – ← phi loop c_i ti [(i', bot), (start, top)]
  setBlock exit

```

Figure 3.1.: Monadic generation of for loop

```

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

```

```

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

```

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

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

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

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

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

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

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

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

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

To loosen this restriction, I implemented SSA recovery pass as part of the quasiquoter. Figure 3.4 and ?? show the quoted and the produced code respectively. <sup>1</sup>

<sup>1</sup>The produced code is after applying the InstCombine optimization as I place more phi-nodes than necessary.

update figure to  
correct ordering  
of bbs

```

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

```

Figure 3.3.: Expanded For Loop

```

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

```

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

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

Figure 3.5.: Expanded For Loop (SSA)





## 4. Implementation

### 4.1. Quasiquoter

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

### 4.2. Extension for-loop

### 4.3. SSA

The standard method to achieve this is to use stack allocated variables instead of registers. The LLVM optimizer then uses the mem2reg pass to transform these into registers, adding the necessary phi-nodes in the process. [2]



## 5. Skeletons

### 5.1. map

### 5.2. fold

### 5.3. scan

[9] My plan is the following:

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

```
32     }  
33 }
```

## 6. Conclusion

### 6.1. Related Work



## A. Listings

```
1 ; Function Attrs: nounwind readonly
2 define double @sum(double* nocapture readonly %a, i32 %length) #0 {
3     %1 = icmp sgt i32 %length, 0
4     br i1 %1, label %.lr.ph.preheader, label %._crit_edge
5
6 .lr.ph.preheader:                                ; preds = %0
7     %2 = add i32 %length, -1
8     %3 = zext i32 %2 to i64
9     %4 = add i64 %3, 1
10    %end.idx = add i64 %3, 1
11    %n.vec = and i64 %4, 8589934576
12    %cmp.zero = icmp eq i64 %n.vec, 0
13    br i1 %cmp.zero, label %middle.block, label %vector.body
14
15 vector.body:                                     ; preds = %.lr.ph.preheader
16     , %vector.body
17     %index = phi i64 [ %index.next, %vector.body ], [ 0, %.lr.ph.preheader ]
18     %vec.phi = phi <4 x double> [ %13, %vector.body ],
19                                     [ zeroinitializer, %.lr.ph.preheader ]
20     %vec.phi6 = phi <4 x double> [ %14, %vector.body ],
21                                     [ zeroinitializer, %.lr.ph.preheader ]
22     %vec.phi7 = phi <4 x double> [ %15, %vector.body ],
23                                     [ zeroinitializer, %.lr.ph.preheader ]
24     %vec.phi8 = phi <4 x double> [ %16, %vector.body ],
25                                     [ zeroinitializer, %.lr.ph.preheader ]
26     %5 = getelementptr double* %a, i64 %index
27     %6 = bitcast double* %5 to <4 x double>*
28     %wide.load = load <4 x double>* %6, align 8
29     %sum22 = or i64 %index, 4
30     %7 = getelementptr double* %a, i64 %sum22
31     %8 = bitcast double* %7 to <4 x double>*
32     %wide.load9 = load <4 x double>* %8, align 8
33     %sum23 = or i64 %index, 8
34     %9 = getelementptr double* %a, i64 %sum23
35     %10 = bitcast double* %9 to <4 x double>*
36     %wide.load10 = load <4 x double>* %10, align 8
37     %sum24 = or i64 %index, 12
38     %11 = getelementptr double* %a, i64 %sum24
39     %12 = bitcast double* %11 to <4 x double>*
40     %wide.load11 = load <4 x double>* %12, align 8
41     %13 = fadd <4 x double> %vec.phi, %wide.load
42     %14 = fadd <4 x double> %vec.phi6, %wide.load9
43     %15 = fadd <4 x double> %vec.phi7, %wide.load10
44     %16 = fadd <4 x double> %vec.phi8, %wide.load11
```

```
44  %index.next = add i64 %index, 16
45  %i7 = icmp eq i64 %index.next, %n.vec
46  br i1 %i7, label %middle.block, label %vector.body, !llvm.loop !0
47
48 middle.block:                                     ; preds = %vector.body, %
    .lr.ph.preheader
49  %resume.val = phi i64 [ 0, %lr.ph.preheader ], [ %n.vec, %vector.body ]
50  %rdx.vec.exit.phi = phi <4 x double> [ zeroinitializer, %lr.ph.preheader
    ],
51                                     [ %i3, %vector.body ]
52  %rdx.vec.exit.phi14 = phi <4 x double> [ zeroinitializer, %lr.ph.preheader
    ],
53                                     [ %i4, %vector.body ]
54  %rdx.vec.exit.phi15 = phi <4 x double> [ zeroinitializer, %lr.ph.preheader
    ],
55                                     [ %i5, %vector.body ]
56  %rdx.vec.exit.phi16 = phi <4 x double> [ zeroinitializer, %lr.ph.preheader
    ],
57                                     [ %i6, %vector.body ]
58  %bin.rdx = fadd <4 x double> %rdx.vec.exit.phi14, %rdx.vec.exit.phi
59  %bin.rdx17 = fadd <4 x double> %rdx.vec.exit.phi15, %bin.rdx
60  %bin.rdx18 = fadd <4 x double> %rdx.vec.exit.phi16, %bin.rdx17
61  %rdx.shuf = shufflevector <4 x double> %bin.rdx18, <4 x double> undef, <4 x
    i32> <i32 2, i32 3, i32 undef, i32 undef>
62  %bin.rdx19 = fadd <4 x double> %bin.rdx18, %rdx.shuf
63  %rdx.shuf20 = shufflevector <4 x double> %bin.rdx19, <4 x double> undef, <4
    x i32> <i32 1, i32 undef, i32 undef, i32 undef>
64  %bin.rdx21 = fadd <4 x double> %bin.rdx19, %rdx.shuf20
65  %i8 = extractelement <4 x double> %bin.rdx21, i32 0
66  %cmp.n = icmp eq i64 %end.idx, %resume.val
67  br i1 %cmp.n, label %._crit_edge, label %lr.ph
68
69 .lr.ph:                                           ; preds = %middle.block, %
    .lr.ph
70  %indvars.iv = phi i64 [ %indvars.iv.next, %lr.ph ], [ %resume.val, %
    middle.block ]
71  %x.01 = phi double [ %i21, %lr.ph ], [ %i8, %middle.block ]
72  %i9 = getelementptr double@ %a, i64 %indvars.iv
73  %i20 = load double@ %i9, align 8
74  %i21 = fadd fast double %x.01, %i20
75  %indvars.iv.next = add i64 %indvars.iv, 1
76  %lftr.wideiv1 = trunc i64 %indvars.iv.next to i32
77  %exitcond2 = icmp eq i32 %lftr.wideiv1, %length
78  br i1 %exitcond2, label %._crit_edge, label %lr.ph, !llvm.loop !3
79
80 ._crit_edge:                                     ; preds = %lr.ph, %
    middle.block, %0
81  %x.0.lcssa = phi double [ 0.000000e+00, %0 ], [ %i21, %lr.ph ], [ %i8, %
    middle.block ]
82  ret double %x.0.lcssa
83 }
84
85 attributes #0 = { nounwind readonly }
86
```



---

```
87 !0 = metadata !{metadata !0, metadata !1, metadata !2}
88 !1 = metadata !{metadata !"llvm.vectorizer.width", i32 1}
89 !2 = metadata !{metadata !"llvm.vectorizer.unroll", i32 1}
90 !3 = metadata !{metadata !3, metadata !1, metadata !2}
```



# References

- [1] Bowen Alpern, Mark N Wegman, and F Kenneth Zadeck. “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>.
- [2] Matthias Braun et al. “Simple and Efficient Construction of Static Single Assignment Form”. In: *Proceedings of the International Conference on Compiler Construction*. Lecture Notes in Computer Science. 2013. URL: <http://pp.info.uni-karlsruhe.de/uploads/publikationen/braun13cc.pdf>.
- [3] Manuel MT Chakravarty 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>.
- [4] Stephen Diehl. *Implementing a JIT Compiled Language with Haskell and LLVM*. Jan. 2014. URL: <http://www.stephendiehl.com/llvm/>.
- [5] Chris Dornan and Simon Marlow. *Alex: A lexical analyser generator for Haskell*. URL: <http://www.haskell.org/alex/>.
- [6] Andy Gill and Simon Marlow. “Happy: the parser generator for Haskell”. In: *University of Glasgow* (1995).
- [7] Timo von Holtz. *llvm-general-quote*. 2014. URL: <https://github.com/tvh/language-llvm-quote>.
- [8] Gabriele Keller 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/repaper.pdf>.
- [9] Richard E Ladner and Michael J Fischer. “Parallel prefix computation”. In: *Journal of the ACM (JACM)* 27.4 (1980), pp. 831–838. URL: <https://engr.smu.edu/~seidel/courses/cse8351/papers/LadnerFischer80.pdf>.
- [10] Chris Arthur Lattner. “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>.
- [11] LLVM Project. *LLVM Language Reference Manual*. URL: <http://llvm.org/docs/LangRef.html>.

- [12] Geoffrey Mainland. “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>.
- [13] Geoffrey Mainland and Manuel MT Chakravarty. *language-c-quote*. URL: <http://hackage.haskell.org/package/language-c-quote>.
- [14] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. “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>.
- [15] Benjamin Scarlet. *llvm-general*. 2013. URL: <http://hackage.haskell.org/package/llvm-general>.