Master Thesis

An LLVM Backend for Accelerate

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

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

Kiel, April 17, 2014

Selbstständigkeitserklärung

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

April 17, 2014

Timo von Holtz

Todo list

Contents

List of Figures

List of Tables

1 Introduction

Write Introduction

2 Technologies

2.1 LLVM

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

2.1.1 LLVM IR

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

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

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

Figure 2.1: sum as a C function

2.2 Accelerate

```
define double @sum(double* %a, i32 %length) {
  %1 = icmp sgt i32 %length, 0
  br i1 %1, label %.lr.ph, label %._crit_edge

.lr.ph:                                           ; preds = %0, %.lr.ph
  %indvars.iv = phi i64 [ %indvars.iv.next, %.lr.ph ], [ 0, %0 ]
  %x.01 = phi double [ %4, %.lr.ph ], [ 0.000000e+00, %0 ]
  %2 = getelementptr double* %a, i64 %indvars.iv
  %3 = load double* %2
  %4 = fadd double %x.01, %3
  %indvars.iv.next = add i64 %indvars.iv, 1
  %lftr.wideiv = trunc i64 %indvars.iv.next to i32
  %exitcond = icmp eq i32 %lftr.wideiv, %length
  br i1 %exitcond, label %._crit_edge, label %.lr.ph

._crit_edge:                                      ; preds = %.lr.ph, %0
  %x.0.lcssa = phi double [ 0.000000e+00, %0 ], [ %4, %.lr.ph ]
  ret double %x.0.lcssa
}
```

Figure 2.2: sum as a LLVM

3 Contributions

3.1 llvm-general-quote

When writing a companyiler 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 general idea is to use a monadic generator to produce the AST on the fly.

Figure 3.1 shows how to implement a simple for loop using monadic generators. As you can tell this is much 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 solution is to use quasiquotation[Mai07] instead. 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.

I implemented *llvm-general-quote*, a quasiquotation library for LLVM. Figure 3.2 shows a for loop using my library.

Figure 3.3 shows the resulting LLVM IR. This is clearly more readable. Furthermore, one can see much more clearly what the produced code will be.

Another advantage of quasiquotation is antiquotation. 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) |]`

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*.

```
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

```
[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
    }
}
|]
```

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

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

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

Figure 3.3: Expanded For Loop

4 Conclusion

4.1 Related Work

Bibliography

[AWZ88] 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`.

[Die14] Stephen Diehl. *Implementing a JIT Compiled Language with Haskell and LLVM*. Jan. 2014. URL: `http://www.stephendiehl.com/llvm/`.

[DM] Chris Dornan and Simon Marlow. *Alex: A lexical analyser generator for Haskell*. URL: `http://www.haskell.org/alex/`.

[GM95] Andy Gill and Simon Marlow. "Happy: the parser generator for Haskell". In: *University of Glasgow* (1995).

[Hol14] Timo von Holtz. *llvm-general-quote*. 2014. URL: `https://github.com/tvh/language-llvm-quote`.

[Lat02] 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`.

[Mai07] 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`.

[MC] Geoffrey Mainland and Manuel MT Chakravarty. *language-c-quote*. URL: `http://hackage.haskell.org/package/language-c-quote`.

[RWZ88] 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`.

[Sca13] Benjamin Scarlet. *llvm-general*. 2013. URL: `http://hackage.haskell.org/package/llvm-general`.