Intermediate Representations
Concepts of Programming Languages CoPL

Malte Skambath
Universität zu Lübeck
malte.skambath@student.uni-luebeck.de

I. Introduction

Developing software in general has the goal to use and run it on real machines or share it with other people who want to use this software on their hardware. As real machines only understand one special machine code based on their hardware-architecture (for example Intel's IA64 CPU-Architecture) programs in a high-level language need to be translated into this machine language.

Obviously we need such kind of compilers for this translation between programming languages and the hardware. But whenever it is possible that our software has to run on different architectures, this requires several compilers.

However developing a whole compiler for each combination of a programming language and hardware-architecture would require much effort and is unpleasant work as the single steps are nearly always the same. Solutions for this problem have already been developed and can be found in practical use. Those solutions, like virtual machines, are abstracting parts of the entire compilation process. For this we need hardware or platform independent abstractions or data structures for programs. Such data structure are called *intermediate representations* (IR) and need to be powerful enough to describe the semantics of a program but also need to be small and simple to be easily translatable into machine code.

It is possible just to use simple programming languages like C as an intermediate representation and use compiler of this language. Nevertheless special languages have been designed to be simpler and allowing more optimizations during all compilation steps.

In this handout we give a short overview about the common types of intermediate representations. At the beginning we give a short and simplified overview about the classical process of compiling. After this we introduce into common machine models and types of intermediate representations used for abstraction in most applications. Finally we present two different implementations of such intermediate representations in section IV.

II. Classical Compile Process

Most classical compilers use about three phases (see Fig. 1) for generating the machine-code for certain hardware out of a given program written in high-level programming language. At the beginning, in the *frontend*-phase, a sequence of tokens, like numbers, strings, or keywords get extracted from the source code. Using this sequence a *parse tree* which represents the syntactic structure of the code will be generated. When a parse tree gets simplified or annotated with special informations it is called an *abstract syntax tree (AST)*. Figure 2 gives an example for such a tree. Those trees are often used in the semantical analysis of the compiling process in which syntactical informations like variable declarations are checked and the control flow graph can be generated. Sometimes they can be converted partially into directed acyclic graphs for detecting redundant use of variables or computations.

The *control flow graph* (Fig. 3) is a directed graph representing all possible paths a program can follow during its execution. Each node contains instructions or is a condition with two outgoing edges. It can be possible to optimize the CFG and finally it can be translated directly into a given assembly or machine language in the last phase of the compiling process. It should be mentioned that already this data structure is kind of an intermediate representation for internal use in a compiler[5], [14].

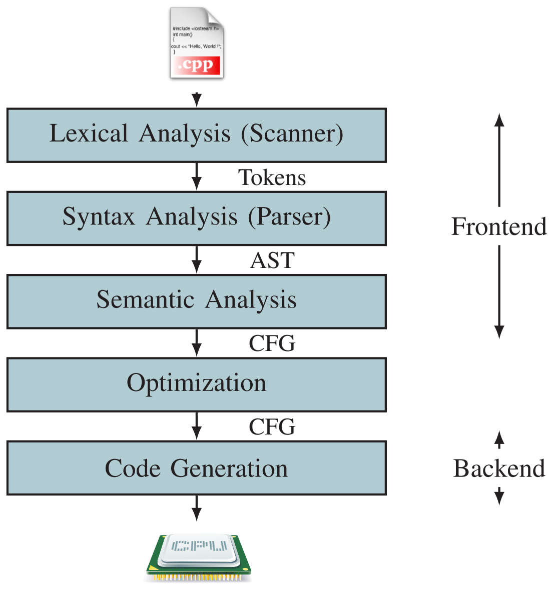

Figure 1: The Architecture of a classical Three-Phase Compiler. In the frontend the control flow of the program will be extracted without dependencies to the target architecture and in the backend the machine-code will be generated for the target architecture.

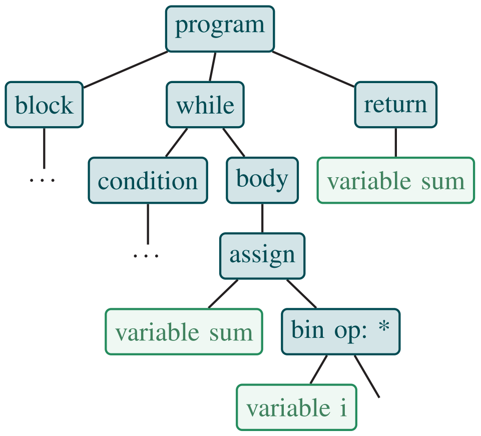

Figure 2: An example for an abstract syntax tree representing the structure of a simple program.

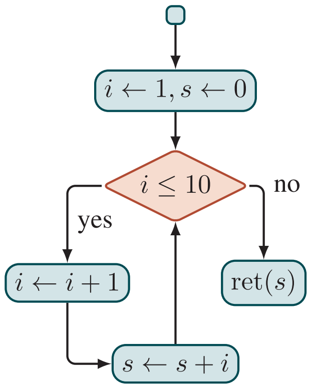

Figure 3: An example for a simple control flow graph to compute the sum $\sum_{i=1}^{10} i$ in a loop.

III. Machine Models

As already mentioned most compilers and especially platform-independent implementations use intermediate representations to represent the control flow graph of a program. Out of this flow they generate machine dependent code in the backend-phase. Thus it is necessary to have a very low-level machine abstraction for this purpose to keep this process simple. This means we need an abstract or *virtual* machine model to have the ability to use assembly-like representation for the control flow which is still flexible enough to be translatable into different real assembly codes. Such a human-readable assembly-language is called *intermediate language*. In real implementations equivalent byte-code-formats with a one-to-one mapping to human-readable assembly languages are used because this prevents additional needless parsing processes between consecutive phases[14].

The differences between assembler codes and the intermediate representations are missing restrictions on hardware like the size or number of available registers or that IR could pass additional informations that may be important for later executed machine-dependent optimizations. This means that hardware-dependent optimization like register-allocation can be separately implemented in the backend.

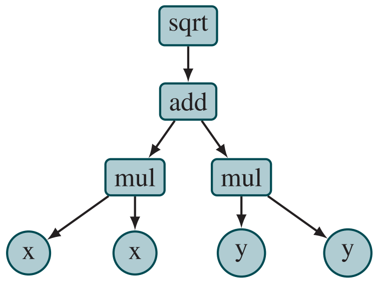

Figure 4: The abstract syntax tree for the expression $\sqrt{x^2 + y^2}$.

In practice there exists two different types of such machine models: stack- and register-based machines. In the following models memory or input-output will not be considered though real implementations as we will see later in section IV need to handle memory and input.output-access.

A. Stack Machines

A stack machine consists of a pushdown-stack and an accumulator or *arithmetic logic unit* (ALU) which is connected with the top elements of this stack. It can put on or remove values from top of its stack and perform operations using the accumulator.

When performing operations like addition or multiplication the top values are taken from the stack, given to the accumulator, and the result is pushed back as new value on top of the stack. This means that for any computation on a stack machine an operations that has to be computed follows pushing or precomputing the values it depends on.

Programs for stack machines are sequences of instructions from a very reduced instruction-set including push and pop operations, some arithmetical functions and also branch instructions to allow non-sequential control flows. So for computations a program has to be written down in reverse polish notation.

While this might be an unfamiliar notation for humans this is a big advantage evaluating expressions given the abstract syntax tree. Because of the operation-nodes in a syntax tree are mostly root nodes for the subtrees of their dependent values the program for a stack machine can directly be generated by traversing the syntax tree in post order. For example computing the euclidean distance for which the expression is presented by the tree in Fig. 4 we would get the following sequence of instruction:

```
push x      ; push the value x
push x      ; push the value x
mul         ; x^2

push y
push y      ; push the value y
mul         ; y^2

add         ; compute x^2+y^2
sqrt
```

Here one can see an advantage of stack based machine models. While register based models always need to reference the involved registers an for stack machines only the operations has to be referenced as the values are on top of the stack. So for stack machines we can write shorter programs for the same computation steps because we don't have to reference target addresses as results are automatically pushed on top of the stack[13].

Programs for a stack machines could be extended with subroutines. When we compute something on the stack with the method we can see that the state of the stack only changes on top where all used values are replaced with the result of a subroutine and there is no reason why the rest of the stack should be touched. This makes it easy to perform subroutines on the same stack. Mostly the calling routine pushes the input variables on the stack before the call.

A big disadvantage of the stack-based model is that most real machines are register machines so the stack-based code has to be translated into register based code and this could mean that special optimization for register machines like register-allocation has to be done in a different step. Some implementations allow use temporal registers or operations on more than the only top values in the stack for this problem or also share special value like the maximum stack-size.

B. Register Machine

A register-based machine has an arbitrary number of memory cells called *registers* that can store values. Operations, like addition, can be done on each (pair) of registers and the value will be stored in another defined register and computations only can be done using registers also if other memories as in real implementations are possible. Each register is available the whole runtime for operations. An Instruction for a register machine can be

- an assignment (`r1 := r2`)
- an unconditional jump (`L1: goto L21`)
- a conditional branch (`L21: if r1 >= 10 goto L42`)
- an arithmetical expression with an assignment (`t1 := 2 * r2`)

1) Three-Address Code (3AC): Because arithmetical expressions can become complex it is a bad idea to allow any complexity or use parenthesis for arithmetical expressions. To prevent problems and simultaneously give the ability for an unique and simple byte-code mapping only one operation with at most two operands is allowed per assignment. Then this register-based assembly language is called *three-address code* (3AC or TAC) because each instruction can be related to at most three registers.

This means we can represent each instruction for example as quadruple like (opcode, regDest, regOp1, regOp2) in byte code. There are much more possibilities for example using triples like (opcode, regOp1, regOp2) where register-address for the result is chosen implicitly by the line number. Note that for such a case a register only may be assigned once which is a strong restriction to the power of a TAC as we will see later.

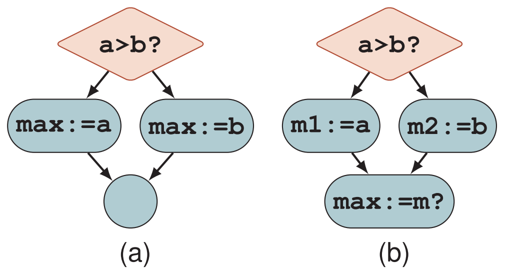

Figure 5: A Branch in the control flow with assignments to the same variable `max` (5a). Assignments in single pathes to the same variable could be replaced to assign two different variables but then the final assignment depends on the chosen path in the CFG (5b)

The following Listing gives us an example of a three-address code for computing $x = \frac{-b+\sqrt{b^2-4ac}}{2a}$:

```
t1 :=  b * b
t2 :=  4 * a
t3 := t2 * c
t4 := t1 - t3
t5 := sqrt(t4)
t6 := 0 - b
t7 := t5 + t6
t8 := 2 * a
t9 := t7 / t8
x  := t9
```

Generating the TAC for a given arithmetical expression is also possible by traversing the abstract syntax tree. Obviously we have to assume that every operation has at most two operands. This means the AST has at most 2 children per node. Now each time we reach a node with operation `<opp>` in post-order we choose a new temporary register t_i and append t_i := `<opp>` (t'_ℓ, t'_r) to the program where t'_r, t'_ℓ will be the temporary registers which contain the result for the subtrees.

2) Static-Single-Assignment: Three-Address Code is good for control flow analysis and other optimization techniques. For example it is possible to compute dependencies or maybe to detect constant propagation and prevent redundant computations. For these reasons it is useful to define another constraint.

A Three-Address Code is in *static-single assignment*-form (SSA) if each register gets assigned only once[14]. This can simplify optimizations for example for register allocation because the life-time of a register and its assigned value during runtime is the same and the dependencies of registers/their values can directly be computed thus the definition-use pairs are explicitly visible in the code.

Transforming general TAC into SSA-form is not trivial. Obviously if there is a register in a sequence of instructions which gets assigned twice we just use a different register since the new assignment and replace each reference to the register in the following part of the sequence until the SSA-condition holds. For example the following sequence of assignments

```
A0:=5; B0:=2; A1:=B0 + A0; B1:=A1 + A1; C0:=B1
```

is semantically equal to

```
A0:=5; B0:=2; A0:=B0 + A0; B0:=A0 + A0; C0:=B0
```

.

and can be generated by just renaming some registers in some statements.

Although this method works for sequences it is wrong in general because the control flow of a program might have branches and the result of a variable could depend on the selected path as you can see this visualized in Fig. 5. We could use two different registers for each branch, but the result for `max` depends on the taken branch so it is impossible to form this into SSA-form directly. One solution for this case it would be store the value in memory (if supported by the register-machine) at assignments in a branch and load it back from the same address after the join. However this could prevent optimizations so in general SSA-form allows a special function ϕ, also called *phony function*[15]. ϕ is defined for the register machine and choses the result value from both operand-registers based on the branch from which this instruction was reached. Our example computation can now be rewritten in SSA-form like:

```
L1: if r_a < r_b then goto L3:
L2:    t_1  := r_a
       goto L4

L3:    t_2  := r_b
       goto L4

L4: max  := phi t_1 [from L2], t_2 [from L3]
```

A temporary register for each branch gets assigned and finally the result depends on the previous instruction of L4.

Note that real machines have no such functions like phi and one has to choose how to abstract the variable selection and what happens with multiple variables or multiple ϕ-instructions. But this is no problem in general because we can and need to transform back from SSA-form before final code-generation. Methods for efficient optimization for SSA are presented in [7].

IV. Implementations

There are several implementations of different kinds of intermediate representations for several virtual machines or compiler engines. In this section we will present two popular implementations. One which use a register-based model and one using a stack-machine.

A. LLVM

The *Low Level Virtual Machine*, known as LLVM, is an infrastructure for compilers. It provides a powerful intermediate representation, such that it is possible to create different frontends for any programming languages. One example for such frontend is clang for C-like programming languages like C, C++, Objective-C or Objective-C++[1].

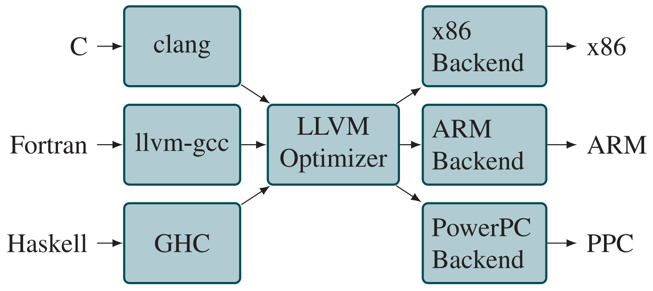

Figure 6: The Three-Phase Architecture of LLVMinfrastructure [10]

Figure 6 shows the architecture of the LLVM infrastructure[4]. For each programming language LLVM only needs a frontend transforming the source code into the LLVM-*Intermediate representation* (LLVM-IR), an SSA-register based representation. With this representation LLVM has the ability to optimize and pass the code also in LLVM-IR to the backend for the specified compiler. This use of IR makes it easy and efficient to implement a new compiler for a new programming language or support a new processor architecture including GPUs[12], [4].

Different tools to perform different optimizations are available and also it is not necessary to directly generate the LLVM-IR from scratch in the frontend as we can use IR-Builder methods in the LLVM-library.

Given a c-program, we can use clang to generate LLVM code in human readable format. Assume we have the following C-Program:

```
#include <stdio.h>
void minmax(int a, int b){
   int max = 0;int min = 0;

   if(a>b)
      {max = a;min = b;}
   else
      {max = b;min = a;}
   printf("␣%d␣<=␣%d␣", min, max);
}
int main(){
   minmax(5,10);
   return 0;
}
```

Using clang with special options like

```
$ clang minmax.c -S -emit-llvm -o - | opt
-S -mem2reg -o -
```

one can get the human readable format of the LLVM intermediate representation. Such that we recieve the produced code for the register-machine:

```
define void @minmax(i32 %a, i32 %b) #0 {
  %1 = icmp sgt i32 %a, %b
  br i1 %1, label %2, label %3

; <label>:2                              ; preds = %0
  br label %4
; <label>:3                        ; preds = %0
  br label %4

11 ; <label>:4                     ; preds = %3, %2
  %max.0 = phi i32 [ %a, %2 ], [ %b, %3 ]
  %min.0 = phi i32 [ %b, %2 ], [ %a, %3 ]
  %5 = call i32 (i8*, ...) @printf(i8*
      getelementptr inbounds ([11 x i8], [11 x
       i8]* @.str, i32 0, i32 0), i32 %min.0,
      i32 %max.0)
  ret void
16 }

declare i32 @printf(i8*, ...) #1

; Function Attrs: nounwind uwtable
21 define i32 @main() #0 {
  call void @minmax(i32 5, i32 10)
  ret i32 0
}
```

Each register is given by a string with % as prefix followed by the register-name. We can see, that clang uses the variable-names from the C-file and generate new registers if required. In line 12 and 13 we can see that ϕ-functions are support using labels for the branches.

Unlike a real machine instruction set we can see that LLVM uses a simple type system. **i8*** for example is a pointer to an octet. Another difference to the general TAC is that the LLVM-IR allows complex function calls and can pass special attributes that could be used by the optimizer or later optimization processes.

While some implementations like the Java virtual machine or the Microsoft.net Framework use a runtime framework on target systems also to provides additional libraries and just-in-time-compilation LLVM does not use this way. But the goal is to allow optimization in each layer of the compiling process including runtime-optimization in the backend. The solution for this is that the intermediate representation is used in each step. So after compiling c-files to .o-files it seems that after this we cannot use IR but generating a .o-file it just stores the IR inside instead of pure machine instructions. And LLVM gives has also the ability to optimize and pass IR-code after linking. This means that a resulting executable file can still contain intermediate which can be compiled and optimized again when the application gets started [4].

However with this architecture it is also no problem just implementing a backend for another high level programming language instead of a low level machine-language and just use LLVM to translate between different languages. This is done for example in a in the emscripten project which uses JavaScript as target language[1].

B. *Common Intermediate Language*

While LLVM provides an infrastructure to develop compilers efficiently by using the LLVM-IR and providing libraries for

[1]Emscripten (http://kripken.github.io/emscripten-site/) is an LLVM-based project allowing the compilation of C or C++ code into JavaScript

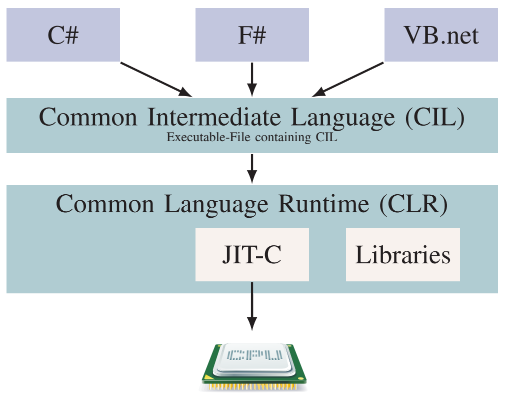
C#
F#
VB.net
Common Intermediate Language (CIL)
Executable-File containing CIL
Common Language Runtime (CLR)
JIT-C
Libraries
CPU

Figure 7: Architecture of the Common Intermediate Infrastructure realised in the Microsoft .net-Framework. The CLR is the implementation of the virtual machine which provides a standard class-library and performs the last compilation-phase.

optimization, code generation and backends for various target systems there is a different way we can use intermediate languages. It is possible to share the software in a non-hardware-dependent format which contains an intermediate representation and move the whole backend phase on target machines. This method requires a special software often called *runtime environment* or *virtual machine* and it is the way the Java Virtual Machine (JVM) and Microsoft.net Framework works.

In the Microsoft .net-Framework applications are created and shared as PE-executable files (file-type: .exe). But those files are no pure PE-files[11]. They contain only a reference to load the main dll of the .net Runtime and payload data which is the intermediate representation that will be compiled just-in-time when the program runs.

The intermediate language for the Microsoft .net-Framework is called *Common Intermediate Language* (CIL) and is part of the ECMA-specification ECMA-335 for the *Common Language Infrastructure* (CLI). This specifies an infrastructure for a runtime environment allowing applications, written in any language, to be executed without the need to rewrite the application for the target platform. The Microsoft .net Framework in this case is Microsoft's implementation of this specification as *Common Language Runtime* and a set of special libraries. Figure 7 shows the basic architecture of the .net-Framework which also contains class library[3], [8].

The CIL is a stack based intermediate language and has much more features than a general stack machine including a strict types and object orientation, and the fact that code can contain meta-informations.

```
public static int sum(int a, int b){
   int res=0;
      for(int i = a; i <= b; i++)
         res += i;
   return res;
}
```

This C# program would be translated into the following CIL-code:

```
.method public static hidebysig
default int32 sum (int32 a, int32 b)  cil
    managed
{
   .maxstack 2
   .locals init (int32  V_0, int32  V_1)
   IL_0000:  ldc.i4.0    //
   IL_0001:  stloc.0      // sum = 0
   IL_0002:  ldarg.0     // load a on the
       stack
   IL_0003:  stloc.1     // store a in first
       var (i=a)
   IL_0004:  br IL_0011  // --+
   IL_0009:  ldloc.0     //   |   <--+
   IL_000a:  ldloc.1     //   |      |
   IL_000b:  add         //   |
   IL_000c:  stloc.0     //   |
   IL_000d:  ldloc.1     //   |
   IL_000e:  ldc.i4.1    //   |
   IL_000f:  add         //   |      .
   IL_0010:  stloc.1     //   |      .
   IL_0011:  ldloc.1     // <-+      .
   IL_0012:  ldarg.1     // load b   |
   IL_0013:  ble IL_0009 // i<=b    -+
   IL_0018:  ldloc.0
   IL_0019:  ret
}
```

With each method-call the *virtual execution engine* (VES) reserves part of the memory on top of an evaluation stack for the arguments and all local visible variables declared at the beginning of a method (see the **`.locals`** instruction in the listing). **`ldloc.`**i and **`stloc.`**i load and store the value of the i-th local variable (**`ldarg`** for argument) onto and from the stack.

As we can see variables are typed. CIL provides a huge set of instructions and also some for arrays, objects and structs. For example using **`newobj`** creates a new object instance and push its this-pointer on the stack. Then loading or storing fields is possible with **`ldfld`** or **`stfld`** which requires also the this-Pointer on the stack before.

Discussion

We just described how and where intermediate representations are used and we can see that the use in general allow us to have clean software-architecture for compilers with multiple layers or steps for the compilation processes. This gives us the possibility to build or extend good infrastructures for developing and optimizing software like LLVM and without the need to solve same problems for different environments. Although this in general simplifies the development of applications the big advantage is that it is not necessary . because one can develop software without regarding to the target system. In addition to that it is also possible to combine modules implemented in different programming languages. Designing new programming or domain specific languages gets faster and easier as developers only now need to implement a frontend which can produce the intermediate representation for the a virtual machine.

While this representations can be powerful we still need to be careful when we implement applications with real-time conditions. It might be difficult to estimate exact runtime when we use optimization methods. In addition if we have programs which should run on parallel systems we have to now if the IR is able to pass special information (or native code binding) or if the IR can handle required features. For example there exists special projects to support OpenMP[2], [6] or also solutions by GPU-producers[12] for LLVM. We can also see the big advantage of the layer architecture as LLVM has already been extended with a separate IR in a new layer for the programming language Swift [9].

References

[1] clang – language compatibility
http://clang.llvm.org/compatibility.html (oct. 2015).

[2] Openmp®/clang
https://clang-omp.github.io/ (oct. 2015).

[3] Overview of the .net framework
https://msdn.microsoft.com/en-us/library/zw4w595w(v=vs.110).aspx (nov. 2015).

[4] Chris Lattner and Vikram Adve. The LLVM Instruction Set and Compilation Strategy. Tech. Report UIUCDCS-R-2002-2292, CS Dept., Univ. of Illinois at Urbana-Champaign, Aug 2002.

[5] Click, C., and Paleczny, M. A simple graph-based intermediate representation. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations* (New York, NY, USA, 1995), IR '95, ACM, pp. 35–49.

[6] Cownie, J. Openmp* support in clang/llvm
http://openmp.org/sc13/OpenMPBoF_LLVM.pdf (oct. 2015).

[7] Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. 13*, 4 (Oct. 1991), 451–490.

[8] European Computer Machinery Association. *Standard ECMA-335: Common Language Infrastructure*, second ed., Dec. 2002.

[9] Groff, J., and Lattner, C. Swift intermediate language - a high level ir complement llvm
http://llvm.org/devmtg/2015-10/slides/GroffLattner-SILHighLevelIR.pdf (nov 2015), 2015.

[10] Lattner, C. The architecture of open source applications – llvm
http://aosabook.org/en/llvm.html (oct. 2015).

[11] Microsoft. Metadata and the pe file structure
https://msdn.microsoft.com/en-us/library/8dkk3ek4(v=vs.100).aspx (nov. 2015).

[12] Nvidia. Cuda llvm compiler
https://developer.nvidia.com/cuda-llvm-compiler (oct. 2015).

[13] Shi, Y., Casey, K., Ertl, M. A., and Gregg, D. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim. 4*, 4 (Jan. 2008), 2:1–2:36.

[14] Stanier, J., and Watson, D. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv. 45*, 3 (July 2013), 26:1–26:27.

[15] Zadeck, K. The development of static single assignment form
http://citi2.rice.edu/WS07/KennethZadeck.pdf (oct. 2015).