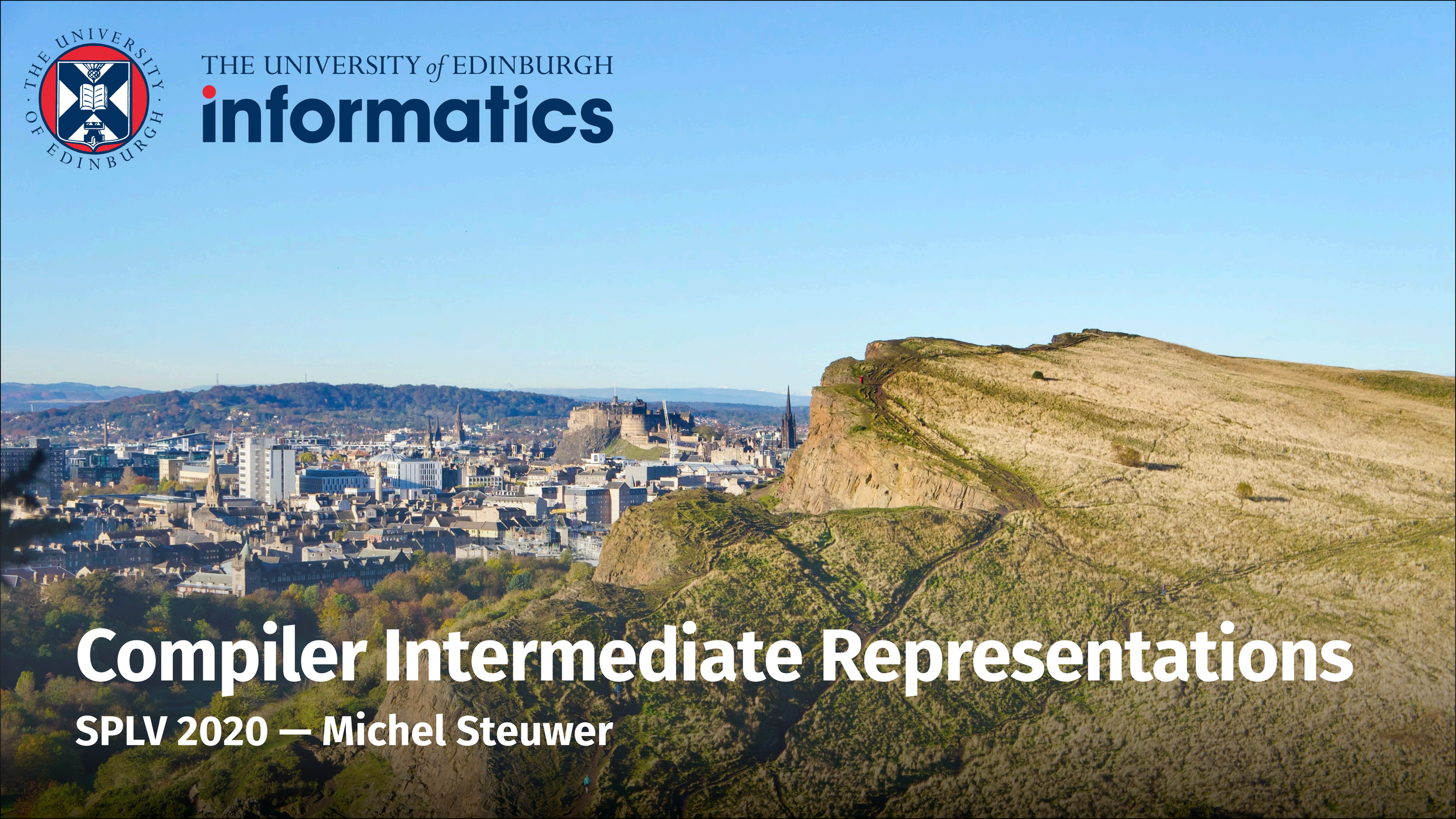


Compiler Intermediate Representations

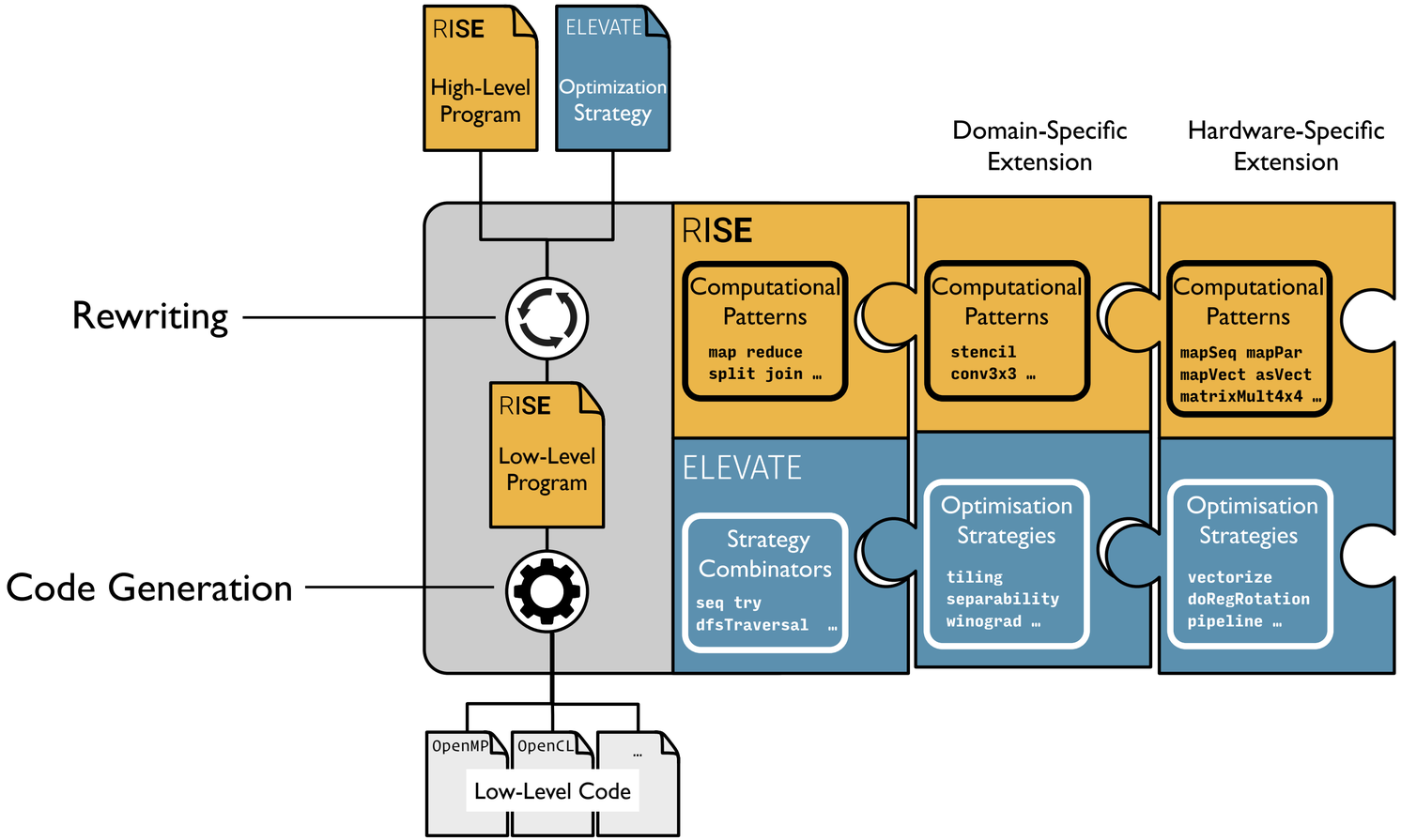
SPLV 2020 — Michel Steuwer

Michel Steuwer

- Since July 2020: Lecturer in Compilers and Runtime Systems at the University *of* Edinburgh
- 2017 - 2020: Lecturer at the University *of* Glasgow
- 2014 - 2017: Post-doc at the University *of* Edinburgh
- 2010 - 2015: PhD studies at the University of Münster in Germany

Main Research Focus:
Parallel Programming, Domain Specific Compilers, Heterogeneous Systems

Current Research: RISE + ELEVATE
A compiler framework for optimising domain-specific applications for specialised hardware



https://rise-lang.org/

https://github.com/rise-lang

https://elevate-lang.org/

https://github.com/elevate-lang

Communications of
the ACM
February 2009

Research and education in compiler technology is more important than ever.

BY MARY HALL, DAVID PADUA, AND KESHAV PINGALI

Compiler Research: The Next 50 Years

WE PRESENT A perspective on the past contributions, current status, and future directions of compiler technology and make four main recommendations in support of a vibrant compiler field in the years to come. These recommendations were drawn from discussions among presenters and attendees at a U.S. National Science Foundation-sponsored Workshop on Future Directions for Compiler Research and

exaggeration to say that compilers and high-level languages are as central to the information age as semiconductor technology.

In the coming decade, 2010 to 2020, compiler research will play a critical role in addressing two of the major challenges facing the overall computer field:

Cost of programming multicore processors. While machine power will continue to grow impressively, increased parallelism, rather than clock rate, will be the driving force in computing in the foreseeable future. This ongoing shift toward parallel architectural paradigms is one of the greatest challenges for the microprocessor and software industries. In 2005, Justin Rattner, chief technology officer of Intel Corporation, said, "We are at the cusp of a transition to multicore, multithreaded architectures, and we still have not demonstrated the ease of programming the move will require..."[3]

Security and reliability of complex software systems. Software systems are increasingly complex, making the need to address defects and security attacks more urgent. The profound economic impact of program defects was discussed in a 2002 study commissioned by the U.S. Department of Commerce National Institute of Standards and Technology (NIST), concluding that program defects "are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6% of the gross domestic product." The 2005 U.S. President's Information Technol-

Communications of the ACM
February 2009

Research and education in compiler technology is more important than ever.

BY MARY HALL, DAVID PADUA, AND KESHAV PINGALI

Compiler Research: The Next 50 Years

WE PRESENT A perspective on the past contributions, current status, and future directions of compiler technology and make four main recommendations in support of a vibrant compiler field in the years to come. These recommendations were drawn from discussions among presenters and attendees at a U.S. National Science Foundation-sponsored Workshop on Future Directions for Compiler Research and

exaggeration to say that compilers and high-level languages are as central to the information age as semiconductor technology.

In the coming decade, 2010 to 2020, compiler research will play a critical role in addressing two of the major challenges facing the overall computer field:

Cost of programming multicore processors. While machine power will continue to grow impressively, increased parallelism, rather than clock rate, will be the driving force in computing in the foreseeable future. This ongoing shift toward parallel architectural paradigms is one of the greatest challenges for the microprocessor and software industries. In 2005, Justin Rattner, chief technology officer of Intel Corporation, said, "We are at the cusp of a transition to multicore, multithreaded architectures, and we still have not demonstrated the ease of programming the move will require..."[3]

Security and reliability of complex software systems. Software systems are increasingly complex, making the need to address defects and security attacks more urgent. The profound economic impact of program defects was discussed in a 2002 study commissioned by the U.S. Department of Commerce National Institute of Standards and Technology (NIST), concluding that program defects "are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6% of the gross domestic product." The 2005 U.S. President's Information Technol-

Communications of the ACM
February 2009

Research and education in compiler technology is more important than ever.

BY MARY HALL, DAVID PADUA, AND KESHAV PINGALI

Compiler

When the field of compiling began in the late 1950s, its focus was limited to the translation of high-level language programs into machine code and to the optimisation of space and time requirements of programs. [...]

The compiler field is increasingly intertwined with other disciplines, including computer architecture, programming languages, formal methods, software engineering, and computer security.

WE PRESENT A perspective on the past contributions, current status, and future directions of compiler technology and make four main recommendations in support of a vibrant compiler field in the years to come. These recommendations were drawn from discussions among presenters and attendees at a U.S. National Science Foundation-sponsored Workshop on Future Directions for Compiler Research and

exaggeration to say that compilers and high-level languages are as central to the information age as semiconductor technology.

In the coming decade, 2010 to 2020, compiler research will play a critical role in addressing two of the major challenges facing the overall computer field:

Cost of programming multicore pro-

economic impact of program defects was discussed in a 2002 study commissioned by the U.S. Department of Commerce National Institute of Standards and Technology (NIST), concluding that program defects "are so prevalent and so detrimental that they cost the U.S. economy an estimated $59.5 billion annually, or about 0.6% of the gross domestic product." The 2005 U.S. President's Information Technol-

Compilers of the past: Fortran

Th first commercial and complete compiler

1957

THE FORTAN AUTOMATIC CODING SYSTEM

BY

J. W. BACKUS, R. J. BEEBER, S. BEST, R. GOLDBERG, L. M. HAIBT, H. L. HERRICK,
R. A. NELSON, D. SAYRE, P. B. SHERIDAN, H. STERN,
I. ZILLER, R. A. HUGHES, AND R. NUTT

FORTRAN Program:

```
1) POLYF(X)=C0+X*(C1+X*(C2+X*C3)).
2) DIMENSION A(1000), B(1000).
3) QMAX=-1.0 E20.
4) DO 5 I=1, 1000.
5) QMAX=MAXF(QMAX, POLYF(A(I)
          +B(I))/POLYF(A(I)-B(I))).
6) STOP.
```

IBM 704 mainframe

THE FORTRAN TRANSLATOR

General Organization of the System

The FORTRAN translator consists of six successive sections, as follows.

Section 1: Reads in and classifies statements. For arithmetic formulas, compiles the object (output) instructions. For nonarithmetic statements including input-output, does a partial compilation, and records the remaining information in tables. All instructions compiled in this section are in the COMPAIL file.

Section 2: Compiles the instructions associated with indexing, which result from DO statements and the occurrence of subscripted variables. These instructions are placed in the COMPDO file.

Section 3: Merges the COMPAIL and COMPDO files into a single file, meanwhile completing the compilation of nonarithmetic statements begun in Section 1. The object program is now complete, but assumes an object machine with a large number of index registers.

Section 4: Carries out an analysis of the flow of the object program, to be used by Section 5.

Section 5: Converts the object program to one which involves only the three index registers of the 704.

Section 6: Assembles the object program, producing a relocatable binary program ready for running. Alsc on demand produces the object program in SHARE symbolic language.

(*Note:* Section 3 is of internal importance only; Section 6 is a fairly conventional assembly program. These sections will be treated only briefly in what follows.)

Compilers of the past: Fortran

The first commercial and complete compiler

1957

THE FORTAN AUTOMATIC CODING SYSTEM

BY

J. W. BACKUS, R. J. BEEBER, S. BEST, R. GOLDBERG, L. M. HAIBT, H. L. HERRICK,
R. A. NELSON, D. SAYRE, P. B. SHERIDAN, H. STERN,
I. ZILLER, R. A. HUGHES, AND R. NUTT

FORTRAN Program:

```
1) POLYF(X)=C0+X*(C1+X*(C2+X*C3)).
2) DIMENSION A(1000), B(1000).
3) QMAX=-1.0 E20.
4) DO 5 I=1, 1000.
5) QMAX=MAXF(QMAX, POLYF(A(I)
          +B(I))/POLYF(A(I)-B(I))).
6) STOP.
```

IBM 704 mainframe

THE FORTRAN TRANSLATOR

General Organization of the System

The FORTRAN translator consists of six successive sections, as follows.

Section 1: Reads in and classifies statements. For arithmetic formulas, compiles the object (output) instructions. For nonarithmetic statements including input-output, does a partial compilation, and records the remaining information in tables. All instructions compiled in this section are in the COMPAIL file.

Section 2: Compiles the instructions associated with indexing, which result from DO statements and the occurrence of subscripted variables. These instructions are placed in the COMPDO file.

Section 3: Merges the COMPAIL and COMPDO files into a single file, meanwhile completing the compilation of nonarithmetic statements begun in Section 1. The object program is now complete, but assumes an object machine with a large number of index registers.

Section 4: Carries out an analysis of the flow of the object program, to be used by Section 5.

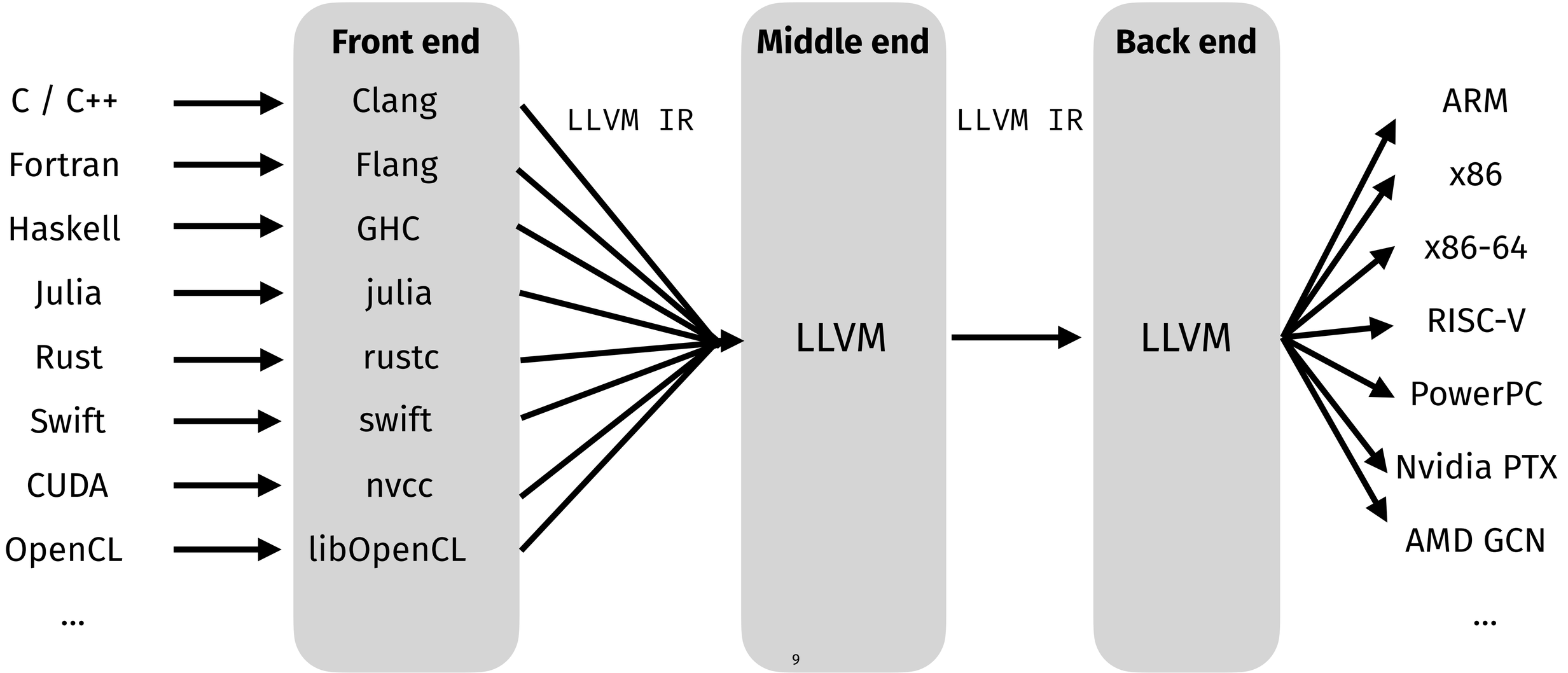
Section 5: Converts the object program to one which involves only the three index registers of the 704.

Section 6: Assembles the object program, producing a relocatable binary program ready for running. Also on demand produces the object program in SHARE symbolic language.

(*Note:* Section 3 is of internal importance only; Section 6 is a fairly conventional assembly program. These sections will be treated only briefly in what follows.)

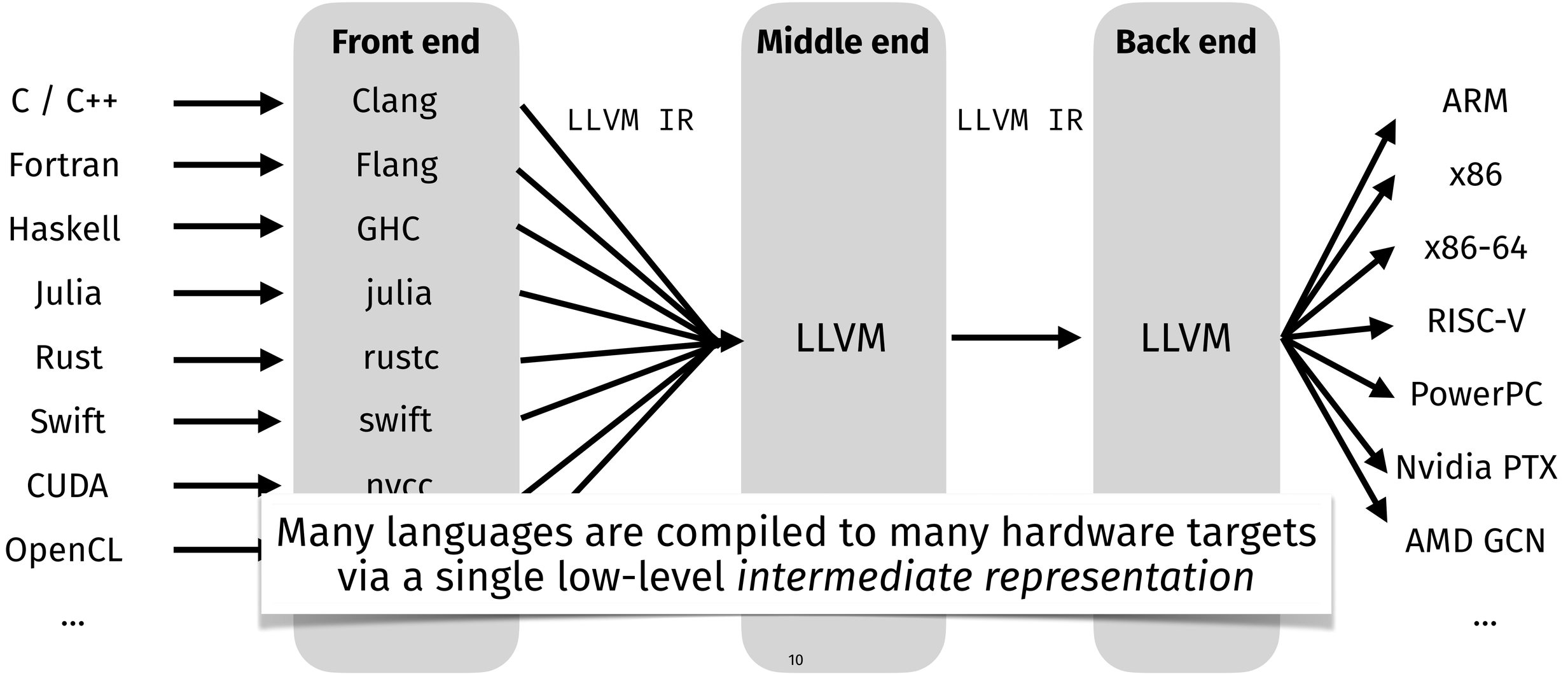
Compilers of the present: LLVM

De-factor standard for industrial compiler infrastructure



Compilers of the present: LLVM

De-factor standard for industrial compiler infrastructure



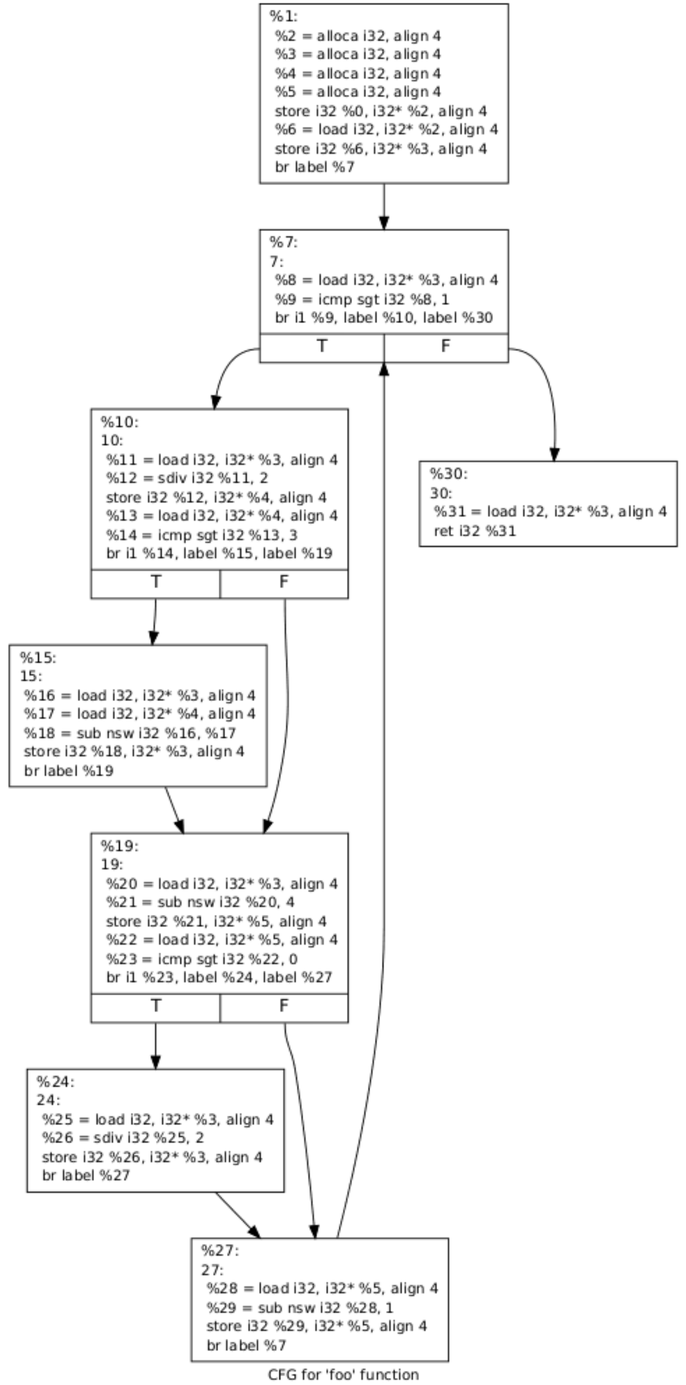
Many languages are compiled to many hardware targets via a single low-level *intermediate representation*

LLVM's Intermediate Representation

" *LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly.*

LLVM Language Reference
https://llvm.org/docs/LangRef.html

```c
int foo(int input) {
  int x,y,z;
  x = input;
  while (x > 1) {
    y = x / 2;
    if (y > 3) x = x - y;
    z = x - 4;
    if (z > 0) x = x / 2;
    z = z-1;
  }
  return x;
}
```



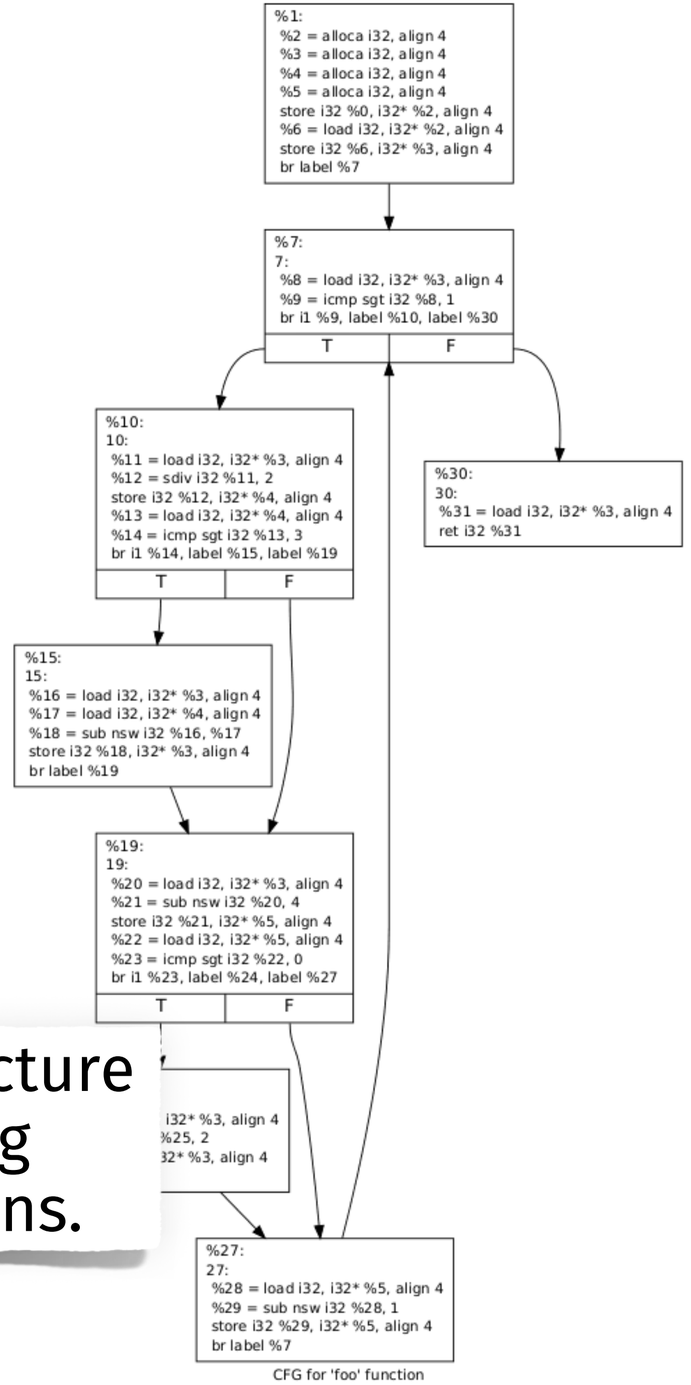
CFG for 'foo' function

LLVM's Intermediate Representation

" *LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing 'all' high-level languages cleanly*

LLVM La
https://llvm

```c
int foo(int input) {
  int x,y,z;
  x = input;
  while (x > 1) {
    y = x / 2;
    if (y > 3) x = x - y;
    z = x - 4;
    if (z > 0) x = x / 2;
    z = z-1;
  }
  return x;
}
```



CFG for 'foo' function

Programs are represented in a graph structure to facilitate data-flow analysis allowing optimisations by re-arranging instructions.

Compilers of the future: ???

One important trend: higher-level intermediate representations

Swift Intermediate Language

A high level IR to complement LLVM

Joe Groff and Chris Lattner

Introducing MIR | Rust Blog

blog.rust-lang.org/2016/04/19/MIR.html

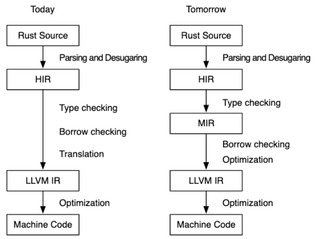
Rust Blog

Rust Install Learn Tools Governance Community

Introducing MIR

Apr. 19, 2016 · Niko Matsakis

We are in the final stages of a grand transformation on the Rust compiler internals. Over the past year or so, we have been steadily working on a plan to change our internal compiler pipeline, as shown here:



Google Developers Blog: XLA

developers.googleblog.com/2017/03/xla-tensorflow-compiled.html

Google Developers

Blog of our latest news, updates, and stories for developers

XLA - TensorFlow, compiled

Monday, March 6, 2017

By the XLA team within Google, in collaboration with the TensorFlow team

One of the design goals and core strengths of TensorFlow is its flexibility. TensorFlow was designed to be a flexible and extensible system for defining arbitrary data flow graphs and executing them efficiently in a distributed manner using heterogenous computing devices (such as CPUs and GPUs).

But flexibility is often at odds with performance. While TensorFlow aims to let you define any kind of data flow graph, it's challenging to make all graphs execute efficiently because TensorFlow optimizes each op separately. When an op with an efficient implementation exists or when each op is a relatively heavyweight operation, all is well; otherwise, the user can still compose this op out of lower-level ops, but this composition is not guaranteed to run in the most efficient way.

This is why we've developed XLA (Accelerated Linear Algebra), a compiler for TensorFlow. XLA uses JIT compilation techniques to analyze the TensorFlow graph created by the user at runtime, specialize it for the actual runtime dimensions and types, fuse multiple ops together and emit efficient native machine code for them - for devices like CPUs, GPUs and custom accelerators (e.g. Google's TPU).

Fusing composable ops for increased performance

Search blog ...

Labels

Archive

Subscribe

Google Developers

YouTube 2M

Follow @googledevs

Visit Google Developers for docs, event info, and more.

So what makes a “good” Intermediate Representation?

acmqueue Intermediate Representation

The increasing significance of intermediate representations in compilers

Fred Chow

Program compilation is a complicated process. A compiler is a software program that translates a high-level source language program into a form ready to execute on a computer. Early in the evolution of compilers, designers introduced IRs (intermediate representations, also commonly called intermediate languages) to manage the complexity of the compilation process. The use of an IR as the compiler’s internal representation of the program enables the compiler to be broken up into multiple phases and components, thus benefiting from modularity.

An IR is any data structure that can represent the program without loss of information so that its execution can be conducted accurately. It serves as the common interface among the compiler components. Since its use is internal to a compiler, each compiler is free to define the form and details of its IR, and its specification needs to be known only to the compiler writers. Its existence can be transient during the compilation process, or it can be output and handled as text or binary files.

THE IMPORTANCE OF IRS TO COMPILERS

An IR should be general so that it is capable of representing programs translated from multiple languages. Compiler writers traditionally refer to the semantic content of programming languages

FIGURE 1 **The Different Levels of Program Representations**

So what makes a “good” Intermediate Representation?

IR DESIGN ATTRIBUTES

In conclusion, here is a summary of the important design attributes of IRs and how they pertain to the two visions discussed here. The first five attributes are shared by both visions.

• **Completeness.** The IR must provide clean representation of all programming language constructs, concepts, and abstractions for accurate execution on computing devices. A good test of this attribute is whether it is easily translatable both to and from popular IRs in use today for various programming languages.

• **Semantic gap.** The semantic gap between the source languages and the IR must be large enough that it is not possible to recover the original source program, in order to protect intellectual property rights. This implies the level of the IR must be low.

• **Hardware neutrality.** The IR must not have built-in assumptions of any special hardware characteristic. Any execution model apparent in the IR should be a reflection of the programming language and not the hardware platform. This will ensure it can be compiled to the widest range of machines, and implies that the level of the IR cannot be too low.

• **Manually programmable.** Programming in IRs is similar to assembly programming. This gives programmers the choice to hand-optimize their code. It is also a convenient feature that helps compiler writers during compiler development. A higher-level IR is usually easier to program.

• **Extensibility.** As programming languages continue to evolve, there will be demands to support new programming paradigms. The IR definition should provide room for extensions without breaking compatibility with earlier versions.

So what makes a “good” Intermediate Representation?

IR DESIGN ATTRIBUTES

In conclusion, here is a summary of the important design attributes of IRs and how they pertain to the two visions discussed here. The first five attributes are shared by both visions.

• **Completeness.** The IR must provide clean representation of all programming language constructs, concepts, and abstractions for accurate execution on computing devices. A good test of this attribute is whether it is easily translatable both to and from popular IRs in use today for various programming languages.

• **Semantic gap.** The semantic gap between the source languages and the IR must be large enough that it is not possible to recover the original source program, in order to protect intellectual property rights. This implies the level of the IR must be low.

??

• **Hardware neutrality.** The IR must not have built-in assumptions of any special hardware characteristic. Any execution model apparent in the IR should be a reflection of the programming language and not the hardware platform. This will ensure it can be compiled to the widest range of machines, and implies that the level of the IR cannot be too low.

??

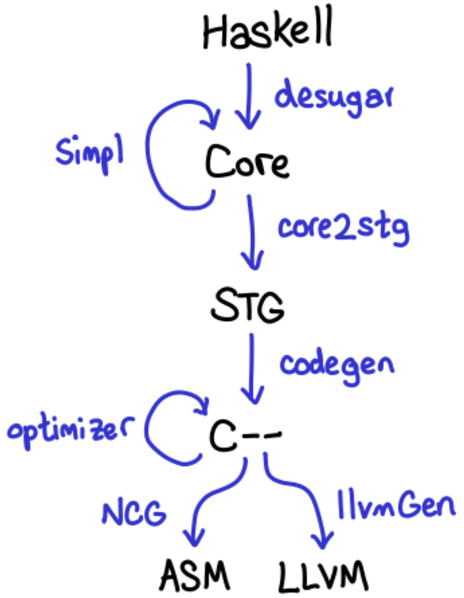
• **Manually programmable.** Programming in IRs is similar to assembly programming. This gives programmers the choice to hand-optimize their code. It is also a convenient feature that helps compiler writers during compiler development. A higher-level IR is usually easier to program.

• **Extensibility.** As programming languages continue to evolve, there will be demands to support new programming paradigms. The IR definition should provide room for extensions without breaking compatibility with earlier versions.

But what about compiling functional languages?

- Functional languages use versions of λ-calculus as intermediate language
- Haskell uses an intermediate language called `Core`
- Its based on the λ-calculus variation *System F*

  System F == *simply typed λ-calculus + polymorphism*



http://blog.ezyang.com/2014/01/so-you-want-to-add-a-new-concurrency-primitive-to-ghc/

Haskell Core

Haskell

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Core

```
map :: forall a b. (a -> b) -> [a] -> [b]
map =
  \ (@ a) (@ b) (f :: a -> b) (xs :: [a]) ->
    case xs of _ {
      []     -> GHC.Types.[] @ b;
      : y ys -> GHC.Types.: @ b (f y) (map @ a @ b f ys)
    }
```

From http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html#(16)

Haskell Core

Haskell

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Core

```
map :: forall a b. (a -> b) -> [a] -> [b]
map =
  \ (@ a) (@ b) (f :: a -> b) (xs :: [a]) ->
    case xs of _ {
      []     -> GHC.Types.[] @ b;
      : y ys -> GHC.Types.: @ b (f y) (map @ a @ b f ys)
    }
```

Programs are represented in the λ-calculus to facilitate optimizations by rewriting.

From http://www.scs.stanford.edu/11au-cs240h/notes/ghc-slides.html#(16)

So who is right? Functional PL or Imperative “Compiler” people?

So who is right? Functional PL or Imperative “Compiler” people?

ACM SIGPLAN
Functional Programming

Editor: *Philip Wadler, Bell Laboratories, Lucent Technologies; wadler@research.bell-labs.com*

SSA is Functional Programming

Andrew W. Appel

Static Single-Assignment (SSA) form is an intermediate language designed to make optimization clean and efficient for imperative-language (Fortran, C) compilers. Lambda-calculus is an intermediate language that makes optimization clean and efficient for functional-language (Scheme, ML, Haskell) compilers. The SSA community draws pictures of graphs with basic blocks and flow edges, and the functional-language community writes lexically nested functions, but (as Richard Kelsey recently pointed out [9]) they're both doing exactly the same thing in different notation.

SSA form. Many dataflow analyses need to find the use-sites of each defined variable or the definition-sites of each variable used in an expression. The *def-use chain* is a data structure that makes this efficient: for each statement in the flow graph, the compiler can keep a list of pointers to all the *use* sites of variables defined there, and a list of pointers to all *definition* sites of the variables used there. But when a variable has N definitions and M uses,

able name for each assignment to the variable. For example, we convert the program at left into the single-assignment program at right. At left, a use of a at any point refers to the most recent definition, so we know where to use a_1, a_2, or a_3, in the program at right.

For a program with no jumps this is easy. But where two control-flow edges join together, carrying different values of some variable i, we must somehow merge the two values. In SSA form this is done by a notational trick, the ϕ-function. In some node with two in-edges, the expression $\phi(a_1, a_2)$ has the value a_1 if we reached this node on the first in-edge, and a_2 if we came in on the second in-edge.

Let's use the following program to illustrate:

$i \leftarrow 1$
$j \leftarrow 1$
$k \leftarrow 0$
while $k < 100$
    if $j < 20$

Maybe both?

So who is right? Functional PL or Imperative “Compiler” people?

Static Single-Assignment (SSA) form is an intermediate language designed to make optimization clean and efficient for imperative-language (Fortran, C) compilers. Lambda-calculus is an intermediate language that makes optimization clean and efficient for functional-language (Scheme, ML, Haskell) compilers. The SSA community draws pictures of graphs with basic blocks and flow edges, and the functional-language community writes lexically nested functions, but (as Richard Kelsey recently pointed out [9]) they're both doing exactly the same thing in different notation.

Outline of Lectures over the week

- **Tuesday:** Functional Intermediate Representations
  - Lambda Calculus and the Lambda Cube
  - Implementation Strategies for System F (ADTs across different PLs)
  - Implementation Strategies for Binders
  - Compiler transformations as rewrite rules
- **Wednesday:** Imperative Intermediate Representations
  - Foundations of Single Static Assignment (SSA)
  - LLVM IR
  - Control-Flow Graphs
  - Data-flow analysis
- **Thursday:** Domain-Specific Intermediate Representations
  - MLIR — a compiler infrastructure for building domain-specific intermediate representations
  - Dataflow graphs — TensorFlow
  - Pattern-based (and functional) — RISE

References

- Compiler Research: The Next 50 Years, *Mary Hall, David Padua, Keshav Pingali* https://dl.acm.org/doi/10.1145/1461928.1461946
- The Fortran Automatic Coding System, *Backus, Beeber, Best, Goldberg, Haibt, Herrick, Nelson, Sayre, Sheridan, Stern, Ziller, Hughes, Nutt* http://www.softwarepreservation.org/projects/FORTRAN/paper/BackusEtAl-FortranAutomaticCodingSystem-1957.pdf
- Intermediate Representation, *Fred Chow* https://dl.acm.org/doi/abs/10.1145/2542661.2544374
- SSA is Functional Programming, *Andrew Appel* https://dl.acm.org/doi/10.1145/278283.278285
- A Correspondence between Continuation Passing Style and Static Single Assignment Form, *Richard Kelsey*, https://dl.acm.org/doi/pdf/10.1145/202530.202532