Compiling Techniques

Lecture 20: Trends in Compiler Design & SSA vs. Functional IRs
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 Education in 2007. As 2007 was the 50th anniversary of IBM's release of the first optimizing compiler, it was a particularly appropriate year to take stock of the status of compiler technology and discuss its future over the next 50 years. Today, compilers and high-level languages are the foundation of the complex and ubiquitous software infrastructure that undergirds the global economy. The powerful and elegant technology in compilers has also been invaluable in other domains (such as hardware synthesis). It is no 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…”

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 Technology Agenda included secure software engineering and software assurance among its top 10 research priorities, concluding with: “Commonly used software engineering practices permit dangerous errors, such as improper handling of buffer overflows, which enable hundreds of attack programs to compromise millions of computers every year. In the future, the Nation [the U.S.] may face even more challenging problems as adversaries—both foreign and domestic—use and adapt these tools.”

Research and education in compiler technology is more important than ever.
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 Education in 2007. As 2007 was the 50th anniversary of IBM’s release of the first optimizing compiler, it was a particularly appropriate year to take stock of the status of compiler technology and discuss its future over the next 50 years. Today, compilers and high-level languages are the foundation of the complex and ubiquitous software infrastructure that undergirds the global economy. The powerful and elegant technology in compilers has also been invaluable in other domains (such as hardware synthesis). It is no 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…”

- **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 Technology Advisory Committee report, *Cyber Security: A Crisis of Prioritization*, included secure software engineering and software assurance among its top 10 research priorities, concluding with: “Commonly used software engineering practices permit dangerous errors, such as improper handling of buffer overflows, which enable hundreds of attack programs to compromise millions of computers every year. In the future, the Nation [the U.S.] may face even more challenging problems as adversaries—both foreign and domestic—learn to exploit these weaknesses.”
Research and education in compiler technology is more important than ever.

BY MARY HALL, DAVID PADUA, AND KESHAV PINGALI

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 Education in 2007. As 2007 was the 50th anniversary of IBM's release of the first optimizing compiler, it was a particularly appropriate year to take stock of the status of compiler technology and discuss its future over the next 50 years. Today, compilers and high-level languages are the foundation of the complex and ubiquitous software infrastructure that undergirds the global economy. The powerful and elegant technology in compilers has also been invaluable in other domains (such as hardware synthesis). It is no 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…"

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 Technology Advisory Committee (PITAC) report Cyber Security: A Crisis of Prioritization included secure software engineering and software assurance among its top 10 research priorities, concluding with: "Commonly used software engineering practices permit dangerous errors, such as improper handling of buffer overflows, which enable hundreds of attack programs to compromise millions of computers every year. In the future, the Nation [the U.S.] may face even more challenging problems as adversaries—both foreign and domestic— seek to exploit software vulnerabilities." 

We present four recommendations to enhance compiler research and education.

Research Recommendation 1: Establish a new field of computer science focused on compiler technology.

Research Recommendation 2: Place increased emphasis on research into compiler and software tools.

Research Recommendation 3: Strengthen the training of graduate students in compiler technology.

Research Recommendation 4: Increase the research and educational emphasis on compiler technology in the undergraduate curriculum.
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, F. B. SHERIDAN, H. STERN,
I. ZILLER, R. A. HUGHES, and R. NUTT

FORTAN 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 FORTAN TRANSLATOR

General Organization of the System

The FORTAN 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

Front end
- C / C++
- Fortran
- Haskell
- Julia
- Rust
- Swift
- CUDA
- OpenCL
- ...

Middle end
- LLVM IR
- LLVM

Back end
- ARM
- x86
- x86-64
- RISC-V
- PowerPC
- Nvidia PTX
- AMD GCN
- ...

Front end tools:
- Clang
- Flang
- GHC
- julia
- rustc
- swift
- nvcc
- libOpenCL
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
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.

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

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

 Circus for "foo" function

1 int foo(int input) {
2     int x, y, z;
3     x = input;
4     while (x > 1) {
5         y = x / 2;
6         if (y > 3) x = x - y;
7         z = x - 4;
8         if (z > 0) x = x / 2;
9         z = z - 1;
10     }
11     return x;
12 }
Compilers of the future: ???
One important trend: higher-level intermediate representations
So what makes a “good” 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
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.
MLIR: Multi-Level Intermediate Representation

A *compiler infrastructure* to define your own intermediate representation to have it interact and integrate with other intermediate representations.

Presenting the work of many, many, people!
What is wrong with existing compilers? — Clang

Huge abstraction gap between C++,... and LLVM IR

Build specialised data structures for C/C++-specific static analysers ...

Better solution: Clang should have a CIR!
High level IRs are not just a good idea for Clang ...

Modern languages pervasively invest in high level IRs

TensorFlow is basically a huge compiler ecosystem

Images by Chris Lattner
But what about compiling functional languages?

- Functional languages use versions of $\lambda$-calculus as intermediate language.
- Haskell uses an intermediate language called Core.
- It's based on the $\lambda$-calculus variation System F.
  
  $$\text{System F} \equiv \text{simply typed } \lambda\text{-calculus} + \text{polymorphism}$$

### Syntax

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>variable</td>
</tr>
<tr>
<td>$\lambda x : T. t$</td>
<td>abstraction</td>
</tr>
<tr>
<td>$tt$</td>
<td>application</td>
</tr>
<tr>
<td>$\lambda X. t$</td>
<td>type abstraction</td>
</tr>
<tr>
<td>$t[T]$</td>
<td>type application</td>
</tr>
</tbody>
</table>

### Values

<table>
<thead>
<tr>
<th>Value</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x : T. t$</td>
<td>abstraction value</td>
</tr>
<tr>
<td>$\lambda X. t$</td>
<td>type abstraction value</td>
</tr>
</tbody>
</table>

### Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>type variable</td>
</tr>
<tr>
<td>$T \rightarrow T$</td>
<td>type of functions</td>
</tr>
<tr>
<td>$\forall X. T$</td>
<td>universal type</td>
</tr>
</tbody>
</table>

### Contexts

<table>
<thead>
<tr>
<th>Context</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\emptyset$</td>
<td>empty context</td>
</tr>
<tr>
<td>$\Gamma, x : T$</td>
<td>term variable binding</td>
</tr>
<tr>
<td>$\Gamma, X$</td>
<td>type variable binding</td>
</tr>
</tbody>
</table>

### Evaluation

- **(E-APP1)**
  - $t_1 \rightarrow t'_1$
  - $t_2 \rightarrow t'_2$
  - $v_1 \cdot t_2 \rightarrow v_1 \cdot t'_2$

- **(E-APPAbs)**
  - $t_1 \rightarrow t'_1$
  - $t_2 \rightarrow [X]T_2$

- **(E-TApp)**
  - $t_1 \rightarrow t'_1$
  - $t_2 \rightarrow [X]T_2$

### Typing

- **(T-VAR)**
  - $x : T \in \Gamma$
  - $\Gamma \vdash x : T$

- **(T-ABS)**
  - $\Gamma, x : T_1 \vdash t_2 : T_2$
  - $\Gamma \vdash \lambda x : T_1. t_2 : T_1 \rightarrow T_2$

- **(T-APP)**
  - $\Gamma \vdash t_1 : T_1 \rightarrow T_12$
  - $\Gamma \vdash t_2 : T_11$
  - $\Gamma \vdash t_1 \cdot t_2 : T_12$

- **(T-TABS)**
  - $\Gamma, X \vdash t_2 : T_2$
  - $\Gamma \vdash \lambda X. t_2 : \forall X. T_2$

- **(T-TAPP)**
  - $\Gamma \vdash t_1 : \forall X. T_12$
  - $\Gamma \vdash t_1 \cdot [T_2] : [X \rightarrow T_2]T_12$

---

Figure 23-1: Polymorphic lambda-calculus (System F)

Types and Programming Languages, B. Pierce
Typed Lambda Calculus

What type system (or logical foundation) do you want?

Terms can bind terms

Types can bind terms

Types can bind types

Polymorphism

Types can bind types

Dependent types

Type Operators

Terms can bind terms

System F

Simply typed $\lambda$-calculus

\[ \lambda \bar{2} \]

\[ \lambda \omega \]

\[ \lambda P \]

\[ \lambda C \]

\[ \lambda P \omega \]

\[ \lambda P 2 \]
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?

Functional Programming

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, an \( O(NM) \) 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 \( \phi \)-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:

\[
\begin{align*}
t &\leftarrow 1 \\
j &\leftarrow 1 \\
k &\leftarrow 0 \\
\text{while } k < 100 \\
&\text{if } j < 20 \\
&\text{then } \phi(a_1, a_2)
\end{align*}
\]
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.
Programs in SSA

\[
\begin{align*}
i &\leftarrow 1 \\
j &\leftarrow 1 \\
k &\leftarrow 0 \\
\textbf{while} \ k < 100 & \\
\textbf{if} \ j < 20 & \\
\ & j \leftarrow i \\
\ & k \leftarrow k + 1 \\
\textbf{else} & \\
\ & j \leftarrow k \\
\ & k \leftarrow k + 2 \\
\textbf{return} \ j & 
\end{align*}
\]

Program

Control Flow Graph

Program in SSA form
SSA is a functional program!

Program in SSA form

Equivalent Functional Program

let \( i_1 = 1 \), \( j_1 = 1 \), \( k_1 = 0 \)
in let function \( f_2(j_2, k_2) = \)
  \[
  \text{if } k_2 < 100 \\
  \text{then let function } f_7(j_4, k_4) = \\
  f_2(j_4, k_4) \\
  \text{in if } j_2 < 20 \\
  \text{then let } j_3 = i_1, k_3 = k_2 + 1 \\
  \text{in } f_7(j_3, k_3) \\
  \text{else let } j_5 = k_2, k_5 = k_2 + 1 \\
  \text{in } f_7(j_5, k_5) \\
  \text{else return } j_2 \\
\]
in \( f_2(j_1, k_1) \)
SSA is a functional program!

Program in SSA form

Basic Blocks are Functions

Control Flow are Function Calls

Equivalent Functional Program

let \( i_1 = 1 \), \( j_1 = 1 \), \( k_1 = 0 \)
in let function \( f_2(j_2, k_2) = \)
if \( k_2 < 100 \)
then let function \( f_7(j_4, k_4) = \)
then let \( j_3 = i_1 \), \( k_3 = k_2 + 1 \)
in \( f_7(j_3, k_3) \)
else let \( j_5 = k_2 \), \( k_5 = k_2 + 1 \)
in \( f_7(j_5, k_5) \)
else return \( j_2 \)
in \( f_2(j_1, k_1) \)
Trends in Compiler Design & SSA vs. Functional IRs

• The role of compilers is evolving
• Appropriate intermediate representations are essential for compilers
• Frameworks (such as MLIR) enable the definition of custom IRs
• Compilers for functional languages usually do not use SSA-based IRs
• There is a correspondence between SSA and functional IRs