CS107: Outlook

Guannan Wei
guannan.wei@tufts.edu
April 21, 2026

Spring 2026

Tufts University

This Week

- Final topics about compilers in practice and research.
- Review for the final exam.
- Final exam: May 7th, 3:30-5:30 pm at JCC 140.

Parsers and Parser Generators

Parsers and Parser Generators

- In the class, we have implemented a recursive-descent parser for simple languages.
- There are many different ways to write parsers, or even generate them automatically from a grammar specification.

Parsers

- Top-down parsing: recognize what is expected to be seen
  - Recursive-descent parsing
  - Many variants of LL (left-to-right, leftmost derivation)
  - Cannot handle left recursion (e.g. $E \rightarrow E + E$) without transformation
- Bottom-up parsing: recognize what has already been seen
  - Many variants of LR (left-to-right, rightmost derivation in reverse)
  - Generally more powerful than top-down parsing

Parser Generators

- Given a grammar specification, automatically generate a parser for it.
- Examples
  - Yacc / Bison for C/C++
  - ANTLR
  - ...

Parser Generators

- Given a grammar specification, automatically generate a parser for it.
- Examples
    - Yacc / Bison for C/C++
    - ANTLR
    - ...
- Less control over error handling and reporting
- Performance tends to be worse than hand-written parser

Compiler Generators

Compiler Generators

- Parser can be generated from a grammar specification, but what about the rest of the compiler?

Compiler Generators

- Parser can be generated from a grammar specification, but what about the rest of the compiler?
- Interpreters and compilers are closely related, and we can mechanically obtain compilers by transforming interpreter!

Partial Evaluation

- Idea: specializing self-applicable partial evaluator
- Futamura projections (Yoshihiko Futamura, 1971)

Systems • Computers • Controls, Vol. 2, No. 5, 1971
Translated from Denshi Tsushin Gakkai Ronbunshi, Vol. 54-C, No. 8, August 1971, pp. 721-728

Partial Evaluation of Computation Process—an Approach to a Compiler-Compiler

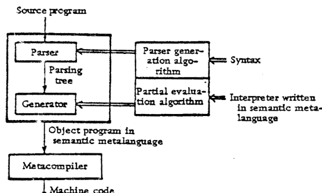
Yoshihiko Futamura, Member

Central Research Laboratory, Hitachi, Ltd., Kokubunji, Tokyo, Japan 185

SUMMARY

This paper reports the relationship between formal description of semantics (i.e., interpreter) of a programming language and an actual compiler. The paper also describes a method to automatically generate an actual compiler from a formal description which is, in some sense, the partial evaluation of a computation process.

The compiler-compiler inspired by this method differs from conventional ones in that the compiler-compiler based on our method can describe



Partial Evaluation

Partial evaluation is the process of evaluating a program with known inputs at compile time, producing a specialized version of the program that is more efficient for those inputs.

```
def pow(x: Int, n: Int): Int =
  if (n == 0) 1 else x * pow(x, n - 1)
```

Partial Evaluation

Partial evaluation is the process of evaluating a program with known inputs at compile time, producing a specialized version of the program that is more efficient for those inputs.

```
def pow(x: Int, n: Int): Int =
  if (n == 0) 1 else x * pow(x, n - 1)
```

Specialize it with n = 3 at compile time (but x is only known at run time):

```
   pow(x, 3)
=> if (3 == 0) 1 else x * pow(x, 2)
=> x * pow(x, 2)
=> x * (if (2 == 0) 1 else x * pow(x, 1))
=> x * (x * (if (1 == 0) 1 else x * pow(x, 0)))
=> x * (x * (x * 1))
```

The partial evaluation result is a *program* `x * (x * (x * 1))` that is more efficient than the original program `pow(x, 3)`.

Partial Evaluation

- Notation: partial evaluator (specializer/mixer) pe; interpreter $eval$ for a source language; $[\![p]\!](x)$ denotes executing program p on x.
- First projection (compilation): $[\![pe]\!](eval, srcPrg) = tgtPrg$
  - such that $[\![eval]\!](srcPrg, input) = [\![tgtPrg]\!](input)$

Partial Evaluation

- Notation: partial evaluator (specializer/mixer) pe; interpreter $eval$ for a source language; $[\![p]\!](x)$ denotes executing program p on x.
- First projection (compilation): $[\![pe]\!](eval, srcPrg) = tgtPrg$
  - such that $[\![eval]\!](srcPrg, input) = [\![tgtPrg]\!](input)$
- Second projection (compiler generation): $[\![pe]\!](pe, eval) = compiler$
  - such that $[\![compiler]\!](srcPrg) = [\![pe]\!](eval, srcPrg) = tgtPrg$

Partial Evaluation

- Notation: partial evaluator (specializer/mixer) pe; interpreter $eval$ for a source language; $[\![p]\!](x)$ denotes executing program p on x.
- First projection (compilation): $[\![pe]\!](eval, srcPrg) = tgtPrg$
  - such that $[\![eval]\!](srcPrg, input) = [\![tgtPrg]\!](input)$
- Second projection (compiler generation): $[\![pe]\!](pe, eval) = compiler$
  - such that $[\![compiler]\!](srcPrg) = [\![pe]\!](eval, srcPrg) = tgtPrg$
- Third projection (compiler-compiler): $[\![pe]\!](pe, pe) = cogen$
  - such that $[\![cogen]\!](eval) = [\![pe]\!](pe, eval) = compiler$

Partial Evaluation

- Notation: partial evaluator (specializer/mixer) pe; interpreter $eval$ for a source language; $[\![p]\!](x)$ denotes executing program p on x.
- First projection (compilation): $[\![pe]\!](eval, srcPrg) = tgtPrg$
  - such that $[\![eval]\!](srcPrg, input) = [\![tgtPrg]\!](input)$
- Second projection (compiler generation): $[\![pe]\!](pe, eval) = compiler$
  - such that $[\![compiler]\!](srcPrg) = [\![pe]\!](eval, srcPrg) = tgtPrg$
- Third projection (compiler-compiler): $[\![pe]\!](pe, pe) = cogen$
  - such that $[\![cogen]\!](eval) = [\![pe]\!](pe, eval) = compiler$
- Fourth projection (self-generation): $[\![cogen]\!](pe) = cogen$

Partial Evaluation

- GraalVM/Truffle (Oracle) is a partial evaluation framework for building compilers from interpreters.
- Support many languages (Python, JavaScript, etc.) running on the JVM with high performance.
- The compiler performs better by removing the interpreter overhead.

Practical Partial Evaluation for High-Performance Dynamic Language Runtimes.
Würthinger et al., PLDI 17

Optimizations

The Phase Ordering Problem

Phase-ordering problem: optimizations can be mutually beneficial, and the order in which they are applied can affect the final performance of the generated code.

- Choose a fixed order
- Iteratively apply optimizations until no more improvement can be made

The Phase Ordering Problem

Phase-ordering problem: optimizations can be mutually beneficial, and the order in which they are applied can affect the final performance of the generated code.

- Choose a fixed order
- Iteratively apply optimizations until no more improvement can be made

But neither guarantees the optimal result!

The Phase Ordering Problem

```
x := 42;
while (...) {
  if (x == 42) {
    f();
  } else {
    g();
    x := x + 1;
  }
}
y := x;
```

```
// How can we optimize it to
// the following code?
x := 42;
while (...) {
  f();
}
y := x;
```

The Phase Ordering Problem

```
x := 42;
while (...) {
  if (x == 42) {
    f();
  } else {
    g();
    x := x + 1;
  }
}
y := x;
```

```
// How can we optimize it to
// the following code?
x := 42;
while (...) {
  f();
}
y := x;
```

- Need constant propagation and dead-code elimination, but ...
  - Constant propagation cannot establish that x is always 10, because of the loop.
  - DCE cannot remove the **else**-branch, because it cannot determine that it is unreachable.

The Phase Ordering Problem

```
x := 42;
while (...) {
  if (x == 42) {
    f();
  } else {
    g();
    x := x + 1;
  }
}
y := x;
```

```
// How can we optimize it to
// the following code?
x := 42;
while (...) {
  f();
}
y := x;
```

- Need constant propagation and dead-code elimination, but ...
  - Constant propagation cannot establish that x is always 10, because of the loop.
  - DCE cannot remove the `else`-branch, because it cannot determine that it is unreachable.
- Each optimization/analysis makes pessimistic assumptions, and combined we get stuck at a local minimum, no matter the order of these optimizations!

Optimal Optimization

Still not a fully solved problem, but there are some promising approaches:

- Super-optimization: search for the optimal sequence of instructions for a given program, often using techniques like theorem proving, brute-force enumeration, or genetic algorithms.
- Equality saturation: represent all possible equivalent programs in a single data structure (e.g., e-graph), and then search for the optimal program within this structure.

Equality Saturation

- `egg` (https://egraphs-good.github.io/) is an e-graph and equality saturation implementation in Rust.
- In e-graph representation, n equalities can be constructed in $O(nlogn)$ time.
- *egg: Fast and extensible equality saturation*. Willsey et al., POPL '20.

Equality Saturation

Defining a simple semi-ring language:

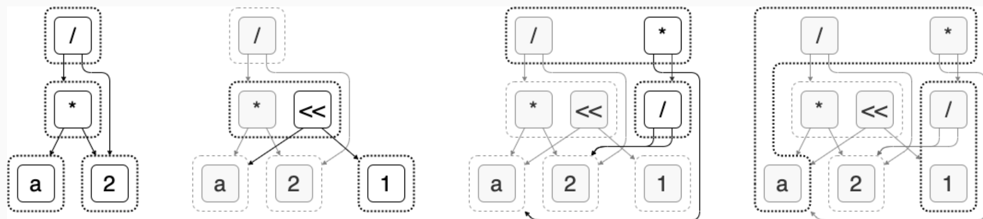
```
define_language! {
    enum SimpleLanguage {
        Num(i32),
        "+" = Add([Id; 2]),
        "*" = Mul([Id; 2]),
        Symbol(Symbol),
    }
}
```

Equality Saturation

Providing the possible algebraic equalities:

```
fn make_rules() -> Vec<Rewrite<SimpleLanguage, ()>> {
    vec![
        rewrite!("commute-add"; "(+ ?a ?b)" => "(+ ?b ?a)"),
        rewrite!("commute-mul"; "(* ?a ?b)" => "(* ?b ?a)"),
        rewrite!("add-0"; "(+ ?a 0)" => "?a"),
        rewrite!("mul-0"; "(* ?a 0)" => "0"),
        rewrite!("mul-1"; "(* ?a 1)" => "?a"),
    ]
}
```

Equality Saturation



- Initial graph represents `(a *2)/2`
- Add equality `x*2 = x <<1`, so `(x <<1)/2` is also capture without forgetting the original program.
- Add equality `(x*y)/z = x*(y/z)`, so `a *(2/2)` is captured.
- Add equality `x/x = 1` and `x*1 = x`, so `a` is captured.

The final representation captures all possible programs under the given equalities, and we can search for the optimal one.

Higher-Order Programs

- We have talked about dataflow analysis and optimizations, but they work on a low-level register-transfer language (RTL) or intermediate representation.
- It doesn't immediately apply to the CPS IR or other higher-levels IRs with first-class functions, or even just function pointers.
- Because functions can be used as data, but they also impact control flow, and the optimizations and analyses need to be aware of both aspects.

Control-Flow Analysis

Control-flow analysis (CFA) is a static analysis technique used to determine the possible control flow of a program, especially in the presence of higher-order functions.

```
def foo(f):
  return f(42) // control-flow question: what is the actual callee of f?

f(lambda x: x + 1) // data-flow question: what is the value passed to f?
```

Control-Flow Analysis

Control-flow analysis on higher-order program is in general undecidable, but can be approximated too.

Compilers for higher-order languages either need to perform control-flow analysis and optimize based on the results, or can convert the program into first-order representation, followed by other optimizations and analyses.

Trustworthy Compilation

Correctness

We built a compiler, but how do we know it is correct? Can we trust it to not introduce bugs or alter the behavior of the program?

Correctness

We built a compiler, but how do we know it is correct? Can we trust it to not introduce bugs or alter the behavior of the program?

Ken Thompson's Hack: it is possible to write a compiler that

- inserts a backdoor into the compiled code (e.g. for a password checker), and
- also inserts backdoor generator when compiling itself, so that the new compiler generated also has the malicious behavior.

Reflections On Trusting Trust, Ken Thompson, Turing Award Lecture, 1984.

Correctness

We built a compiler, but how do we know it is correct? Can we trust it to not introduce bugs or alter the behavior of the program?

- We can test it, and it indeed can find a lot bugs.
- Testing is not easy as you may thought – we need to generate programs satisfying certain properties (e.g., grammar, type system, etc.).
- Testing in practice could often find many bugs.

Correctness

For the past three years, we have used Csmith to discover bugs in C compilers. Our results are perhaps surprising in their extent: to date, we have found and reported more than 325 bugs in mainstream C compilers including GCC, LLVM, and commercial tools. Figure 1 shows a representative example. Every compiler that we have tested, including several that are routinely used to compile safety-critical embedded systems, has been crashed and also shown to silently miscompile valid inputs. As measured by the responses to our bug

Finding and Understanding Bugs in C Compilers. Yang et al., PLDI 11

Correctness

For the past three years, we have used Csmith to discover bugs in C compilers. Our results are perhaps surprising in their extent: to date, we have found and reported more than 325 bugs in mainstream C compilers including GCC, LLVM, and commercial tools. Figure 1 shows a representative example. Every compiler that we have tested, including several that are routinely used to compile safety-critical embedded systems, has been crashed and also shown to silently miscompile valid inputs. As measured by the responses to our bug

Finding and Understanding Bugs in C Compilers. Yang et al., PLDI 11

- But testing is not exhaustive, and cannot guarantee the absence of bugs.
- Can we have higher assurance and trust in compilers?

Establish Trustworthy in Compilation

- Translation validation: validate the correctness of each compilation by checking that the source and target programs are semantically equivalent.
  - $\forall S, C, Validate(S, C) = true \Rightarrow S \approx C$
  - No need to trust the compiler, but need to verify the validator.

Establish Trustworthy in Compilation

- Translation validation: validate the correctness of each compilation by checking that the source and target programs are semantically equivalent.
  - $\forall S, C, Validate(S, C) = true \Rightarrow S \approx C$
  - No need to trust the compiler, but need to verify the validator.
- Verified compilation: prove the correctness of the compiler itself, so that any program compiled by it is guaranteed to be correct.
  - $\forall S, C, Compile(S) = \mathsf{OK}(C) \Rightarrow S \approx C$
  - The compiler is verified and can be trusted when it produces $\mathsf{OK}(C)$.

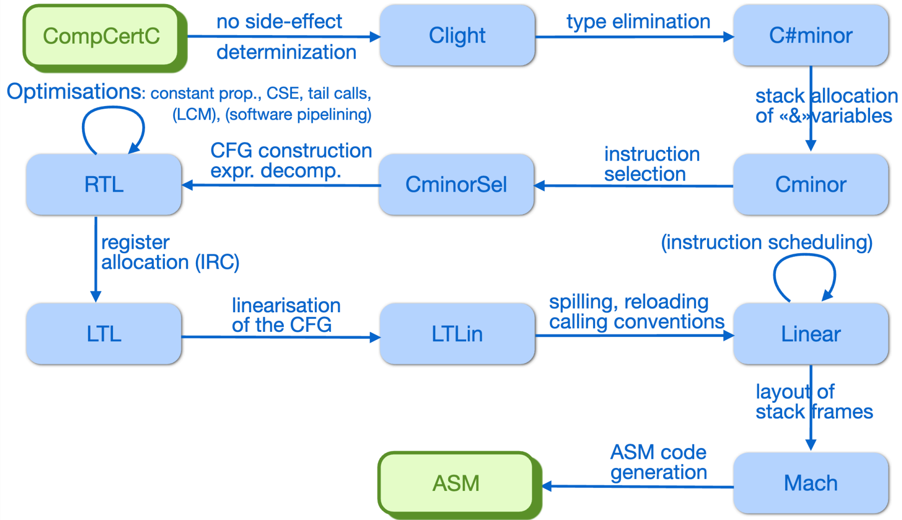
Verified Compilation: CompCert

A prominent example of verified compiler is CompCert (https://compcert.org/), a realistic C compiler that has been formally verified to be correct.

Semantic preservation theorem: For all source programs S and compiler-generated code C, if the compiler, applied to the source S, produces the code C, without reporting a compile-time error, then the observable behavior of C is one of the possible observable behaviors of S.

Formal verification of a realistic compiler. Leroy. Comm. ACM, 2009.

Verified Compilation: CompCert



Verified Compilation: CompCert

- Each pass is implemented and verified using a proof assistant language Rocq/Coq ($\sim$ 170k lines of code+proof).

A tiny example in Rocq:

```
Fixpoint compile (e : exp) : prog :=
  match e with
    | Const n => iConst n :: nil
    | Binop b e1 e2 => compile e2 ++ compile e1 ++ iBinop b :: nil
  end.

Theorem compile_correct : forall e,
  progDenote (compile e) nil = Some (expDenote e :: nil).
Proof.
  ...
```

Verified Compilation: CompCert

- Support C99/C11 with a few minor exceptions.
- Decent performance of generated code (80-90% of GCC).
- Can generate several architectures (PowerPC, ARM, RISC-V and x86).
- ACM Software System Award 2021
- ACM SIGPLAN Programming Languages Software award 2022

Verified Compilation: CompCert

- Support C99/C11 with a few minor exceptions.
- Decent performance of generated code (80-90% of GCC).
- Can generate several architectures (PowerPC, ARM, RISC-V and x86).
- ACM Software System Award 2021
- ACM SIGPLAN Programming Languages Software award 2022
- If you want to learn about verification and proof assistants, consider taking my CS150 Special Topics in Spring 27!

Verified Compilation: CompCert

The striking thing about our CompCert results is that the middle-end bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users.

Finding and Understanding Bugs in C Compilers. Yang et al., PLDI 11

Compilers Everywhere

Compilers Everywhere

- Deep learning (neural networks/LLMs) compilers
- Data analytics and query compilation
- Digital fabrication such as 3D printers
- ...

The problems we talked about in the class exist in all these domains, and the techniques we have learned can be applied to them as well.

Deep Learning Compilers

- A deep learning compiler transforms high-level neural network programs into efficient executable (training/inference) code for hardware.
- Eventually, the code is executed as just matrix/linear algebra operations.
- Examples: PyTorch, TensorFlow, JAX, etc.

Deep Learning Compilers

- A deep learning compiler transforms high-level neural network programs into efficient executable (training/inference) code for hardware.
- Eventually, the code is executed as just matrix/linear algebra operations.
- Examples: PyTorch, TensorFlow, JAX, etc.
- Optimizations: operator fusion, rewriting, scheduling, parallelization, etc.

Deep Learning Compilers

- A deep learning compiler transforms high-level neural network programs into efficient executable (training/inference) code for hardware.
- Eventually, the code is executed as just matrix/linear algebra operations.
- Examples: PyTorch, TensorFlow, JAX, etc.
- Optimizations: operator fusion, rewriting, scheduling, parallelization, etc.
- Diverse and heterogeneous hardware:
  - CPU (vectorization, SIMD)
  - GPU (massive parallelism)
  - Specialized accelerators, TPU, etc.

Query Compilation

- Transform a high-level declarative query into an efficient executable plan.
- Execution speed becomes more critical as memory becomes cheaper and faster, and the bottleneck shifts from IO to computation.
- Users write queries in SQL (what to compute, not how)
- Pipeline: SQL $\rightarrow$ Logical plan $\rightarrow$ Physical plan $\rightarrow$ Machine-level execution

Query Compilation

High-level SQL query:

```
SELECT COUNT(*)
FROM R, S
WHERE R.name == "R1"
  AND R.sid == S.rid
```

Intermediate query plan:

```
AggOp(
  HashJoinOp(
    SelectOp(R, "name", EQ, "R1"),
    S, "sid", "rid"),
  COUNT)
```

Query Compilation

Eventually generating efficient executable code:

```
val MR: Array[R] = ... // Indexed R on sid
/* Actual Query Processing */
var count = 0
for (s <- S) {
  val r = MR(s.rid)
  if (r.name == "R1") {
    if(r.sid == s.rid) {
      count += 1
    }
  }
}
count
```

How to Architect a Query Compiler. Shaikhha et al., SIGMOD '16.

Digital Fabrication

- How do a user design and fabricate a 3D object?
- They don't directly manipulate the motor on the 3D printer!

Digital Fabrication

- How do a user design and fabricate a 3D object?
- They don't directly manipulate the motor on the 3D printer!
- Create a high-level design using a CAD software
- Compile the design into surface mesh representation (e.g., STL file)
- Slice the mesh and generate G-code for the printer to execute

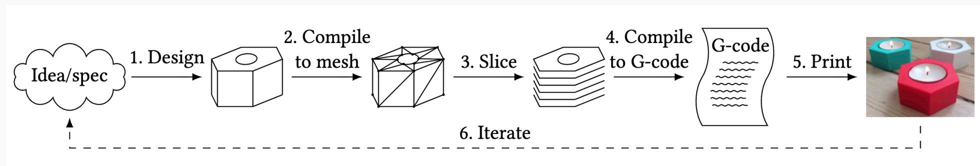


Image from *Functional Programming for Compiling and Decompiling Computer-Aided Design*, Nandi et al., ICFP '18

Surface mesh representation is the triangulation of the object surface.

```
facet normal 0.866025 0.5 0
  outer loop
    vertex 1.0 0.0 1
    vertex 0.5 0.866025 0
    vertex 0.5 0.866025 1
  endloop
endfacet
```

This is the "intermediate representation" for the 3D object!

G-code is the instruction to manipulate the control motors of the 3D printer, and is also used in CNC machines and other digital fabrication tools.

```
M204 S500
;TYPE:Skirt/Brim
G1 F1200
G1 X88.913 Y105.56 E.02426
G1 X89.599 Y105.287 E.02315
```

This is the “assembly language” for the 3D printer!

Takeaways

- Compiler is not only a translator, but automates the process to eliminate abstractions!
- If you want to explore research in compilers/PL, talk to me about potential research projects!