Correctness of Compilers

Viktor Kunčak

Plan

Are there obviously incorrect compilers in practice?

How to define correctness: per run vs in general

Compiler correctness projects

Expression language correctness

Proving correctness using Stainless verifier for Scala

Incorrect Compilers

```
int i,j, x;
int main() {
  int *p;

  p = &x;
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i;

  p = &i;
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i; // k => 2*k+1
  j = i;

  if (x > 0) p = &i; else p = &x; // x > 0 is true
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i;

  printf("x=%2d,j=%2d,i=%2d\n",x,j,i);

}
```

Incorrect Compilers

```
int i,j, x;
int main() {
  int *p;

  p = &x;
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i;

  p = &i;
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i; // k => 2*k+1
  j = i;

  if (x > 0) p = &i; else p = &x; // x > 0 is true
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i;

  printf("x=%2d,j=%2d,i=%2d\n",x,j,i); // default: x=45, j=15, i=15

}
```

Incorrect Compilers

```
int i,j, x;
int main() {
  int *p;

  p = &x;
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i;

  p = &i;
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i; // k => 2*k+1
  j = i;

  if (x > 0) p = &i; else p = &x; // x > 0 is true
  *p = 0; for (i = 0; i < 10; i++) *p = (*p) + i;

  printf("x=%2d,j=%2d,i=%2d\n",x,j,i); // default:  x=45, j=15, i=15
                                        // -xO4:  x=45, j=15, i=45
}
```

Explanation (“Credible Compilation” by Darko Marinov, 2000)

Compiler incorrectly concluded that it is allowed to optimize the last loop in the same way that it is allowed to optimize the first loop:

```
*p = 0; for (i = 0; i < 10; i++) *p = (*p) + i;

===>

t = *p;
for (i = 0; i < 10; i++)
  t = t + i;
*p = t;
```

The optimization is helpful as `t` will likely end up only in a register.

The optimization is only valid when one can be certain that `p` is not stored in the same place in memory as (aliased to) `i`. Compiler incorrectly checked this condition.

Program enumeration for rigorous compiler testing, PLDI 2017

"In less than six months, our approach has led to 217 confirmed GCC/Clang bug reports, 119 of which have already been fixed, and the majority are long latent despite extensive prior testing efforts. Our SPE algorithm also provides six orders of magnitude reduction. Moreover, in three weeks, our technique has found 29 CompCert crashing bugs and 42 bugs in two Scala optimizing compilers."

— Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. SIGPLAN Not. 52, 6 (June 2017), 347–361. `https://doi.org/10.1145/3140587.3062379`

Program enumeration for rigorous compiler testing, PLDI 2017

"In less than six months, our approach has led to 217 confirmed GCC/Clang bug reports, 119 of which have already been fixed, and the majority are long latent despite extensive prior testing efforts. Our SPE algorithm also provides six orders of magnitude reduction. Moreover, in three weeks, our technique has found 29 CompCert crashing bugs and 42 bugs in two Scala optimizing compilers."

— Qirun Zhang, Chengnian Sun, and Zhendong Su. 2017. Skeletal program enumeration for rigorous compiler testing. SIGPLAN Not. 52, 6 (June 2017), 347–361. `https://doi.org/10.1145/3140587.3062379`

If we find bugs, we still need to worry:

1. who will fix them and when?
2. did we find them all?

Formal Verification (see CS-550 next year!)

Use mathematical proof to establish that program satisfies mathematical properties.

Correctness beyond "it works on our test suite": it works for *all possible inputs*:

$$\forall i, o.\;\; run(program, i) = o \;\rightarrow\; correct(o, i)$$

Formal Verification (see CS-550 next year!)

Use mathematical proof to establish that program satisfies mathematical properties.

Correctness beyond “it works on our test suite”: it works for *all possible inputs*:

$$\forall i, o.\;\; run(program, i) = o \;\rightarrow\; correct(o, i)$$

Applying formal verification to correctness of compilers is interesting:

- the problem is difficult (compiler must work on all programs and their inputs)
- we can agree on the specification
  (arguably easier than on the specification of a web browser or a flight simulator)

Certified Compilers vs Compilers Proven Correct

Certifying compiler: compiler generates, for each input, a proof that that the resulting program is correct.

- for some input programs, cerifying compiler can fail (e.g., generate a program for which it is not sure it is correct, as before),
- whenever it succeeds with the proof, we know that the resulting program formally corresponds to starting program
- compiler can be written in any language, we can even retrofit existing compilers to generate such proofs, or have certification passes that guess the proofs
  George C. Necula. 2000. Translation validation for an optimizing compiler. In Proceedings of the ACM SIGPLAN 2000 PLDI. 83–94. *`https://doi.org/10.1145/349299.349314`*
- a special case of *proof carrying code*: low-level programs that can be checked to satisfy key correctness properties

Verified compiler: there is a proof that when the compiler succeeds, the generated program is correct. Our focus today!

Examples of Verified Compilers

- CompCert: `https://compcert.org/`
  "The main result of the project is the CompCert C verified compiler, a high-assurance compiler for almost all of the C language (ISO C 2011), generating efficient code for the PowerPC, ARM, RISC-V and x86 processors."
- CakeML: `https://cakeml.org/`
  CakeML is a functional programming language and an ecosystem of proofs and tools built around the language. The ecosystem includes a proven-correct compiler that can bootstrap itself.
- Bedrock2: `https://github.com/mit-plv/bedrock2`
  "ongoing work on a low-level systems programming language. One piece of the puzzle is a verified compiler targeting RISC-V"

When Was a Program Compiled Correctly?

We would like the compiled program to behave *as we expect*.

When Was a Program Compiled Correctly?

We would like the compiled program to behave *as we expect*.

What defines expectations?

When Was a Program Compiled Correctly?

We would like the compiled program to behave *as we expect*.

What defines expectations?
Source code! How to make the expectations precise?

When Was a Program Compiled Correctly?

We would like the compiled program to behave *as we expect*.

What defines expectations?
Source code! How to make the expectations precise?
Approach:

1. define a simple and clear interpreter: need not be efficient (or even executable: operational semantics), so it is much simpler
2. prove that the interpreter would give same result as the compiler, *for all program inputs*:

$$\forall i, o.\ \ run(compile(program), i) = o \ \rightarrow\ interpret(program, i) = o$$

When Was a Program Compiled Correctly?

We would like the compiled program to behave *as we expect*.

What defines expectations?
Source code! How to make the expectations precise?
Approach:

1. define a simple and clear interpreter: need not be efficient (or even executable: operational semantics), so it is much simpler
2. prove that the interpreter would give same result as the compiler, *for all program inputs*:

$$\forall i, o.\ \ run(compile(program), i) = o \ \rightarrow\ interpret(program, i) = o$$

Let us see simplest examples of such proofs!

Correctness of Compilation to Stack Machine

Consider a tiny expression language and a stack machine.

Goal: show that running compiled program behaves same as evaluating the expression.

We present the tiny compiler using the subset of Scala supported by the Stainless verification system.

Expressions

```
abstract class Binary {
 def apply(x: Int, y:Int): Int
}
case class Plus() extends Binary {
 def apply(x: Int, y:Int) = x + y
}
case class Times() extends Binary {
 def apply(x: Int, y:Int) = x * y
}
sealed abstract class Expr
case class Var(str: String) extends Expr
case class ConstExpr(c: Int) extends Expr
case class BinExpr(left: Expr, b: Binary, right: Expr) extends Expr

def expr1 = BinExpr(BinExpr(Var("x"), Times(), Var("y")),
   Plus(), BinExpr(BinExpr(Var("y"), Times(), Var("z")),
   Plus(), BinExpr(Var("x"), Times(), Var("z"))))
```

Interpreter is Simple

```
type Value = Int
type Env = String => Value

def eval(en: Env)(expr: Expr): Value = {
  expr match {
    case ConstExpr(c) => c
    case Var(s) => en(s)
    case BinExpr(left, b, right) =>
      b(eval(en)(left), eval(en)(right)) // b has an apply
  }
}

def env1 : Env = { (v: String) =>
  if (v == "x") 3 else if (v == "y") 4
  else if (v == " z") 5 else 0
}

def resEval1 = eval(env1)(expr1) // what does it compute?
```

Our Little Stack Machine

```
sealed abstract class Instr
case class Push(c: Int) extends Instr
case class Load(s: String) extends Instr
case class BinOp(b: Binary) extends Instr
type Bytecodes = List[Instr]
type Stack = List[Value]
def push(v: Value, s: Stack): Stack = Cons(v, s)

def run(bs: Bytecodes)(en: Env, stack: Stack): Option[Stack] =
  bs match {
    case Nil() => Some(stack)
    case Cons(Push(c), bs1) => run(bs1)(en, push(c, stack))
    case Cons(Load(s), bs1) => run(bs1)(en, push(en(s), stack))
    case Cons(BinOp(b), bs1) => stack match {
      case Cons(v1, Cons(v2, stack1)) =>
        run(bs1)(en, push(b(v2, v1), stack1))
      case _ => None[Stack]()
    }
  }
```

Our Compiler

```
def compile(expr: Expr): Bytecodes = {
  expr match {
    case ConstExpr(c) => List[Instr](Push(c))
    case Var(s) => List[Instr](Load(s))
    case BinExpr(left, b, right) =>
      compile(left) ++ (compile(right) ++ List[Instr](BinOp(b)))
  }
}

def bytecodes1 = compile(expr1)

def resCompile1 = run(bytecodes1)(env1, List[Value]())
```

Correctness Statement

```
run (compile(expr) ++ bs) (en, inStack) ==

run (bs) (en, push(eval(en)(expr), inStack))
```

In particular, for empty bs and stack:

```
run (compile(expr)) (en, emptyStack) ==

Some(push(eval(en)(expr), emptyStack))
```

Proof: Well-Founded Induction on Expression and Stack

```
// THEOREM: for all expr, bs, stck
run(compile(expr) ++ bs)(en, stck) === run(bs)(en, push(eval(en)(expr), stck))

// case expr == BinExpr(left, b, right)

val op = List[Instr](BinOp(b))
val vLeft = eval(en)(left)
val vRight = eval(en)(right)
val midStack = push(vLeft, inStack)

run(compile(expr) ++ bs)(en, inStack) ===
run((compile(left) ++ (compile(right) ++ op)) ++ bs)(en, inStack) === (assoc)
run(compile(left) ++ (compile(right) ++ (op ++ bs)))(en, inStack) === (IH)
run(compile(right) ++ (op ++ bs))(en, midStack) === (IH)
run(op ++ bs)(en, push(vRight, midStack)) ===
run(bs)(en, push(b(vLeft, vRight), inStack)) ===
run(bs)(en, push(eval(en)(expr), inStack))
```

Another type of proof: type soundness

Our language only had numbers.

If we had an interpreter, we would like to type check the program and know that the interpreter does not crash, and we can be sure that compiler does not need to worry about the behavior for type incorrect code.

Hence, we can try to formally verify type soundness.
I present examples for simple language without recursion or loops, just expressions.

Expression Language with Integers and Booleans

```
sealed trait Expr
case class IntConst(c: BigInt) extends Expr
case class BoolConst(b: Boolean) extends Expr
case class ApplyOperator(e1: Expr, op: Operator, e2: Expr) extends Exp
case class If(condition: Expr, thenBranch: Expr, elseBranch: Expr) ext

// Several example operators
sealed trait Operator
case class PlusOp() extends Operator
case class MinusOp() extends Operator
case class EqOp() extends Operator
case class StrictAndOp() extends Operator // evaluates both
```

Defensive vs "Bold" Interpreter

Defensive (`eval`):

```
case If(c, e1, e2) =>
  eval(c) match
    case Some(BoolVal(b)) =>
      if b then eval(e1) else eval(e2)
    case _ => None[Value]()
```

Bold interpreter (`evalTyped`) would just do:

```
case If(c, e1, e2) =>
  evalTyped(c, BoolType()) match
    case BoolVal(b) =>
      if b then evalTyped(e1, t1) else evalTyped(e2, t2)
```

We will convince stainless that, when the program type checks, then the bold interpreter does not crash and gives the same result.

Stating Type Soundness in Stainless

```
def eval(e: Expr): Option[Value] = // `None` if e not type correct
 ...

// Infer the type of an expression. `None` means it does not type check
def typeOf(e: Expr): Option[TypeExpr] =

// Require expr to have a type
// `t` is a witness that the expression was well typed
def evalTyped(e: Expr, @ghost t: TypeExpr): Value = {
  require(typeOf(e) == Some(t))

  ...
}.ensuring(res => valTypeOf(res) == t && eval(e) == Some(res))
// result has same type as input, equals value of "safe" eval

// This means that well-typed expressions do not get stuck.
```