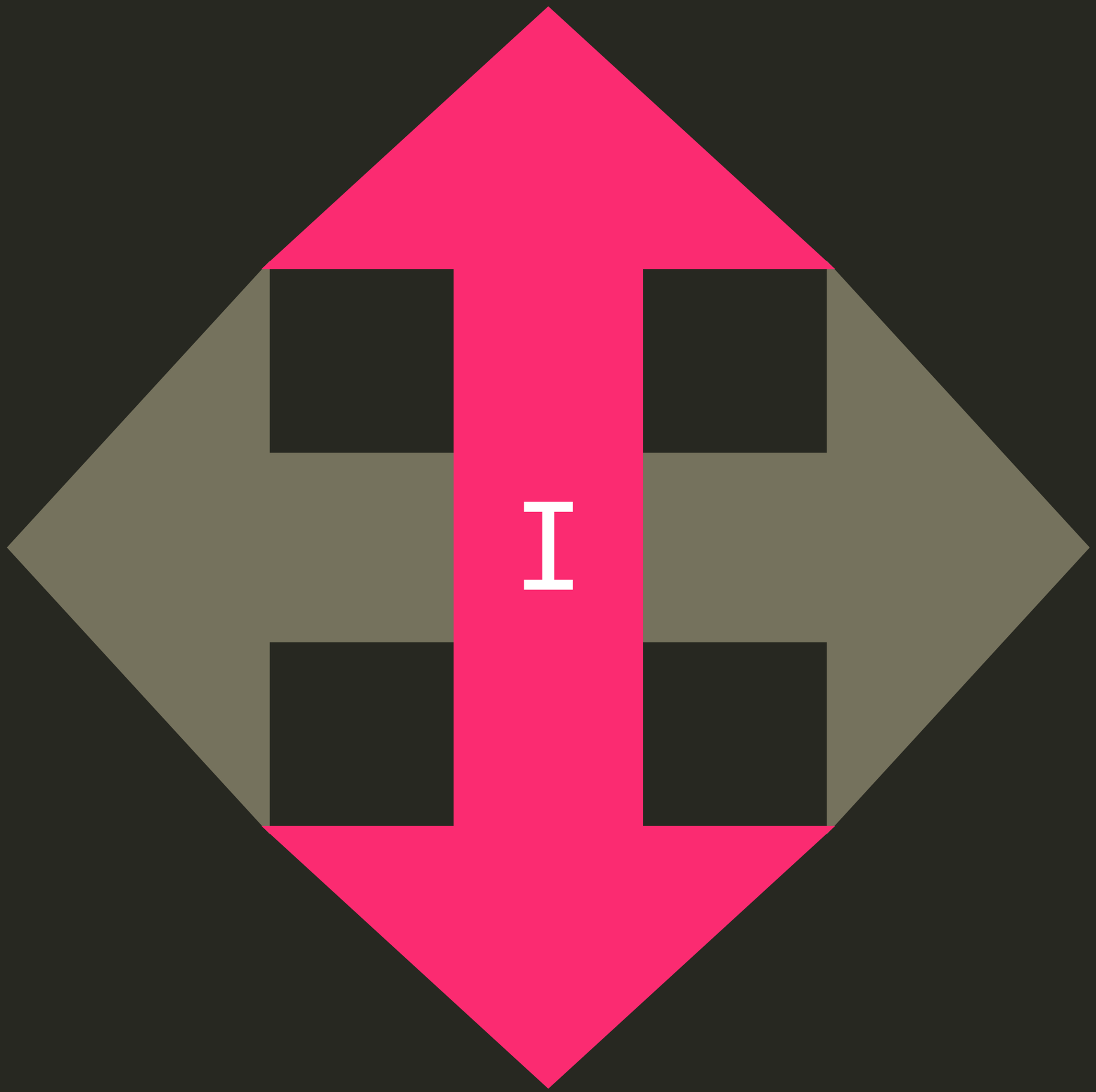
Source Code Manipulation

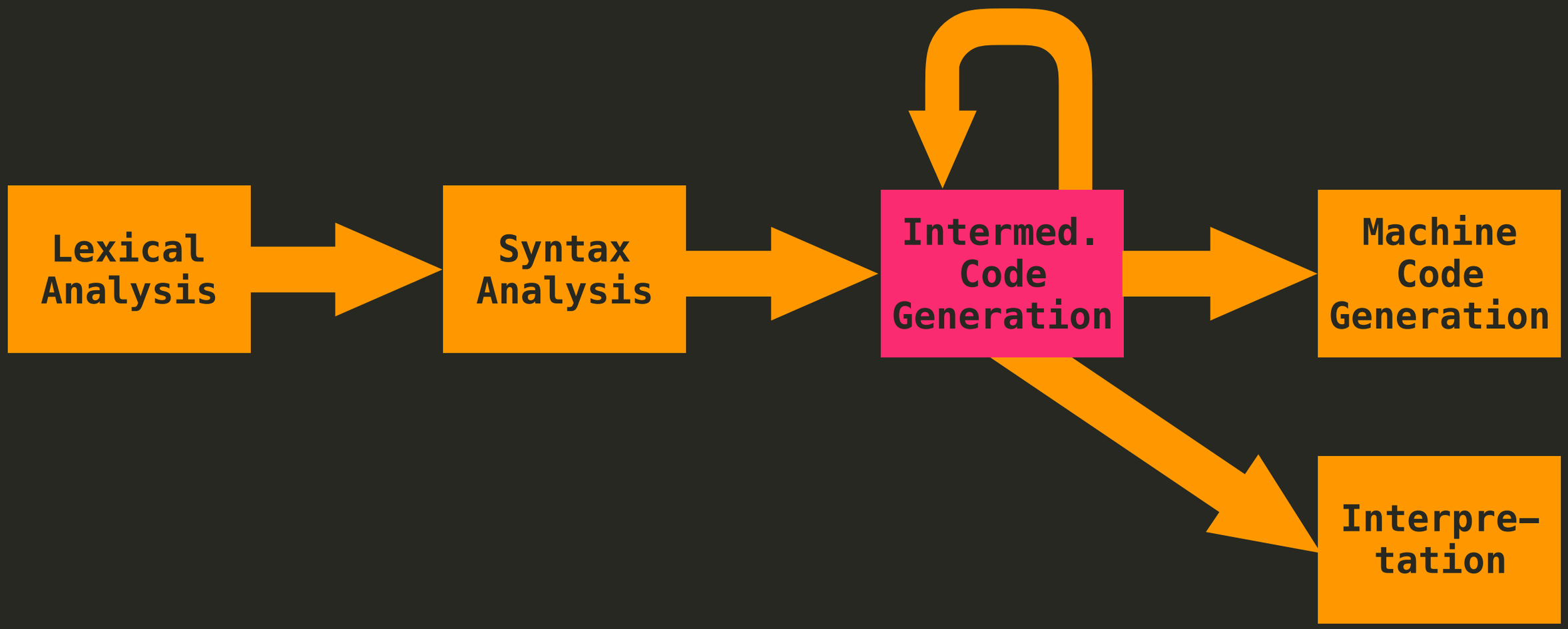
Dr. Vadim Zaytsev aka @grammarware
UvA, MSc SE, 30 November 2015

Roadmap

W44	Introduction	V.Zaytsev
W45	Metaprogramming	J.Vinju
W46	Reverse Engineering	V.Zaytsev
W47	Software Analytics	M.Bruntink
W48	Clone Management	M.Bruntink
W49	Source Code Manipulation	V.Zaytsev
W50	Legacy and Renovation	TBA
W51	Conclusion	V.Zaytsev



Compiler



D.Grune, K.v.Reeuwijk, H.E.Bal, C.J.H.Jacobs, K.Langendoen, Modern Compiler Design, 2ed, 2012, p. 300.

Generated code

* Preferably
  * avoid any evolution
  * regenerate on sync
* Possibly
  * bidirectional link
* Properties:
  * correctness, speed, size, energy…

F.Ferreira, B.Pientka, Bidirectional Elaboration of Dependently Typed Programs, PPDP 2014.

Supercompilation

* History
* partial evaluation (1964, L.A.Lombardi & B.Raphael?)
* supercompilation (1966, Valentin Turchin)
* local simplification (1975–)
* subgoal abstraction (1975)
* symbolic execution (1976, James C. King)
* mixed computation (1977, Andrei Ershov)
* Futamura projections (1983, Yoshihiko Futamura)
* abstract interpretation (1977, P. & R. Cousot)
* . . .

Supercompilation

* Given is F(X,Y); find G(X) = F(X,z)
  * partial application (currying)
  * partial evaluation (residual)
* Also covers:
  * lazy evaluation
  * theorem proving
  * problem solving

Supercompilation

```
* map f $ map g xs


* map (f . g) xs
```

Supercompilation

```
* let ones = 1:ones
  in map (\ x -> x + 1) ones
```

```
* let twos = 2:twos
  in twos
```

Supercompilation

```
* sum x = case x of
            [] -> 0
            x:xs -> x + sum xs
  range i n = case i>n of
                True -> []
                False -> i:range (i+1) n
  main n = sum (range 0 n)

* main2 i n = if i>n
              then 0
              else i + main2 (i+1) n
  main n = main2 1 n
```

Generative SE

* Program generator
  * a program that produces programs
  * in a high-level language

* Structured program generation
  * any generated program should type check
  * (it will be before running anyway)
  * (any error is a bug in a generator)

Everyone's Doing It!

```
* sqlProg = "SELECT name FROM" +
    tableName + "WHERE id = " + id;
```

```
* sqlProg = new SelectStmt(
        new Column("name"),
        table,
        new WhereClause(new Column("id"),
        id));
```

Everyone's Doing It!

* template<int X, int Y>
  struct Adder
    { enum { result = X + Y }; };

* aspect S
  {
    declare parents:
    Car implements Serializable;
  }

Everyone's Doing It!

```
* expr = `[7 + i];
```

```
* stmt = `[
    if (i > 0) return #[expr];
  ];
```

Staging

* Scala, MetaML, MetaOCaml, …
* Explicit delaying of computation
  * quote
  * unquote
  * run/eval

MetaOCaml

```
let even n = (n mod 2) = 0;;
let square x = x * x;;
let rec powerS n x =
  if n = 0 then .<1>.
  else if even n
       then .<square .~(powerS (n/2) x)>.
       else .<.~x *  .~(powerS (n-1) x)>.;;

let power5 = !. .<fun x -> .~(powerS 5 .<x>.)>.;;
```

Scala

```
def powerS (n : Rep[Int], x : Int) : Rep[Int] = {
  if (n == 0) 1
  else if (n % 2 == 0) {
    val result = powerS(n/2, x)
    result * result
  }
  else x * powerS(n-1, x)
}

def powerTest(n : Rep[Int]) : Rep[Int] = powerS(n, 5)
```

Java + MorphJ

```
class LogMe<class X> extends X {
  <R,A*>[m] for ( public R m(A) : X.methods )
  public R m (A a) {
    R result = super.m(a);
    System.out.println(result);
    return result;
  }
}
```

Java + MorphJ

```
class Listify<Subj> {
  Subj ref;
  Listify(Subj s) {ref = s;}

  <R,A>[m] for (public R m(A): Subj.methods)
  public R m (List<A> a) {
   // … call m for all elements
  }
}
```

Java + SafeGen

```
#defgen MakeDelegator ( input(Class c) => !Abstract(c) ) {
  #foreach( Class c : input(c) ) {
    public class Delegator extends #[c] {
      #foreach(Method m : MethodOf(m, c) & !Private(m)) {
        #[m.Modifiers] #[m.Type] #[m] ( #[m.Formals] ) {
          return super.#[m](#[m.ArgNames]);
        }
      }
    }
  }
}
```

Pigs from Sausages

* Interactive disassembly
  * IDA Pro
* Tool-independent
  * Dava, Boomerang, dcc
* Compiler-specific
  * javac: Mocha, Jad, Jasmin,
    Wingdis, SourceAgain

Decompilation uses

* recover lost source code
* adapt to another platform
* check security-critical code
* find malware
* inspect vulnerabilities
* learn algorithms & data formats

Mike Van Emmerik, http://www.program-transformation.org/Transform/WhyDecompilation

Decompilation

* Load binary code into virtual memory
* Parse / disassemble
* Recognise compilation patterns
* Build control flow graph
* Perform data flow analysis
* Perform control flow analysis
* Restructure intermediate result
* Generate high-level code

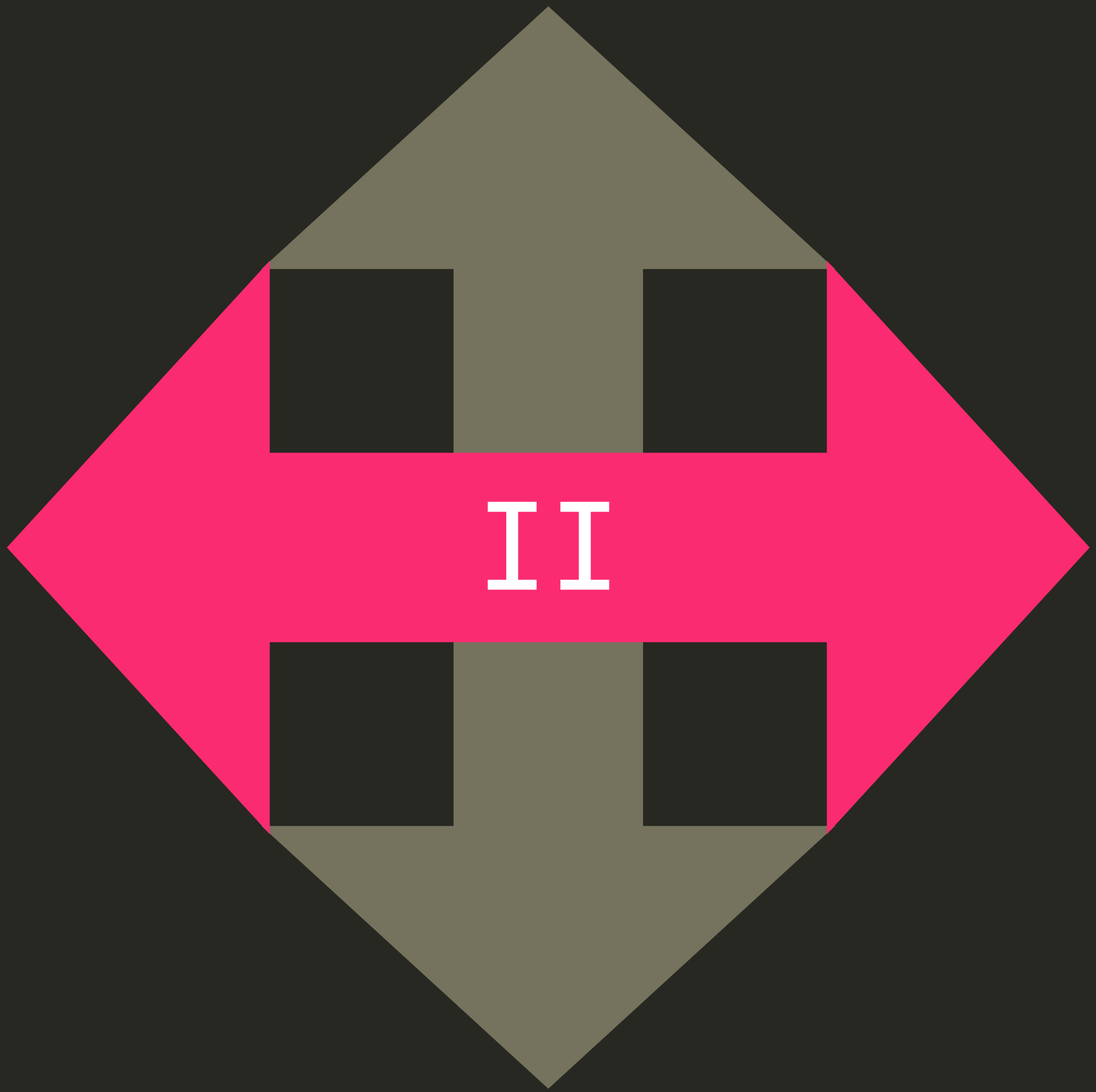
C.Cifuentes, K.J.Gough, Decompilation of Binary Programs, SPE 25(7), 1995

Disasm advice

* Do not underestimate debuggers
  * ptrace, gdb, windbg
  * winice, softice, linice
  * vmware, dosbox, bochs, xen, parallels
* Obfuscation & deobfuscation
  * elfcrypt, upx, burneye, shiva
* Learn system software
* Beware of anti-hacking hacks

Part I: Conclusion

* Compilation and code generation
* Supercompilation
* Generative programming
  * morphing as improved generics
  * staging as guided evaluation
* You want meta-type safety

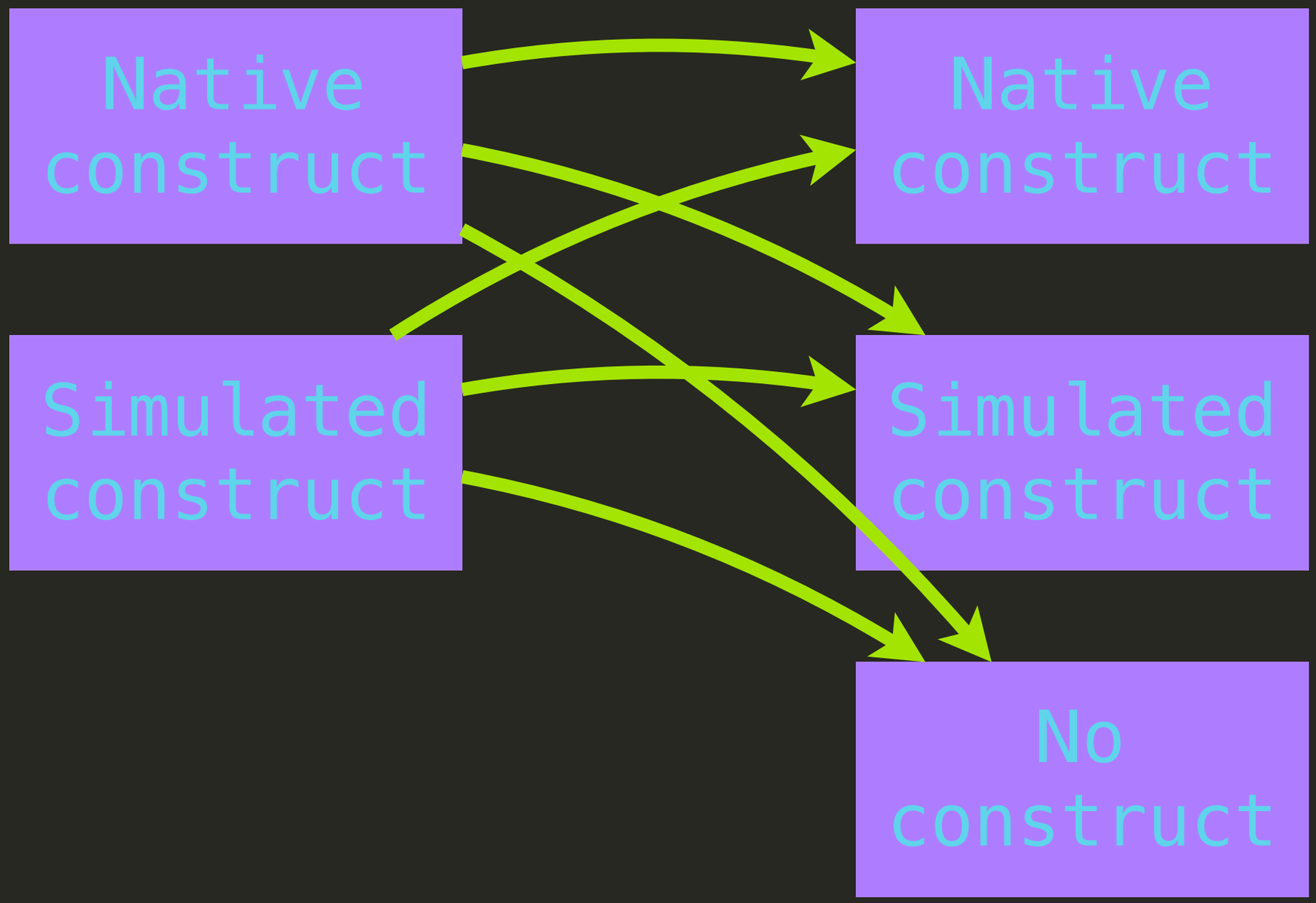


Language Conversion

* Everybody lies.
* Syntax swap is NEVER a solution.
  * not even OS/VS COBOL to VS COBOL II
* Wrapping is NOT a solution!
* Component wrapping COULD be a solution for a while.
* *Two wrongs make a right, almost.*

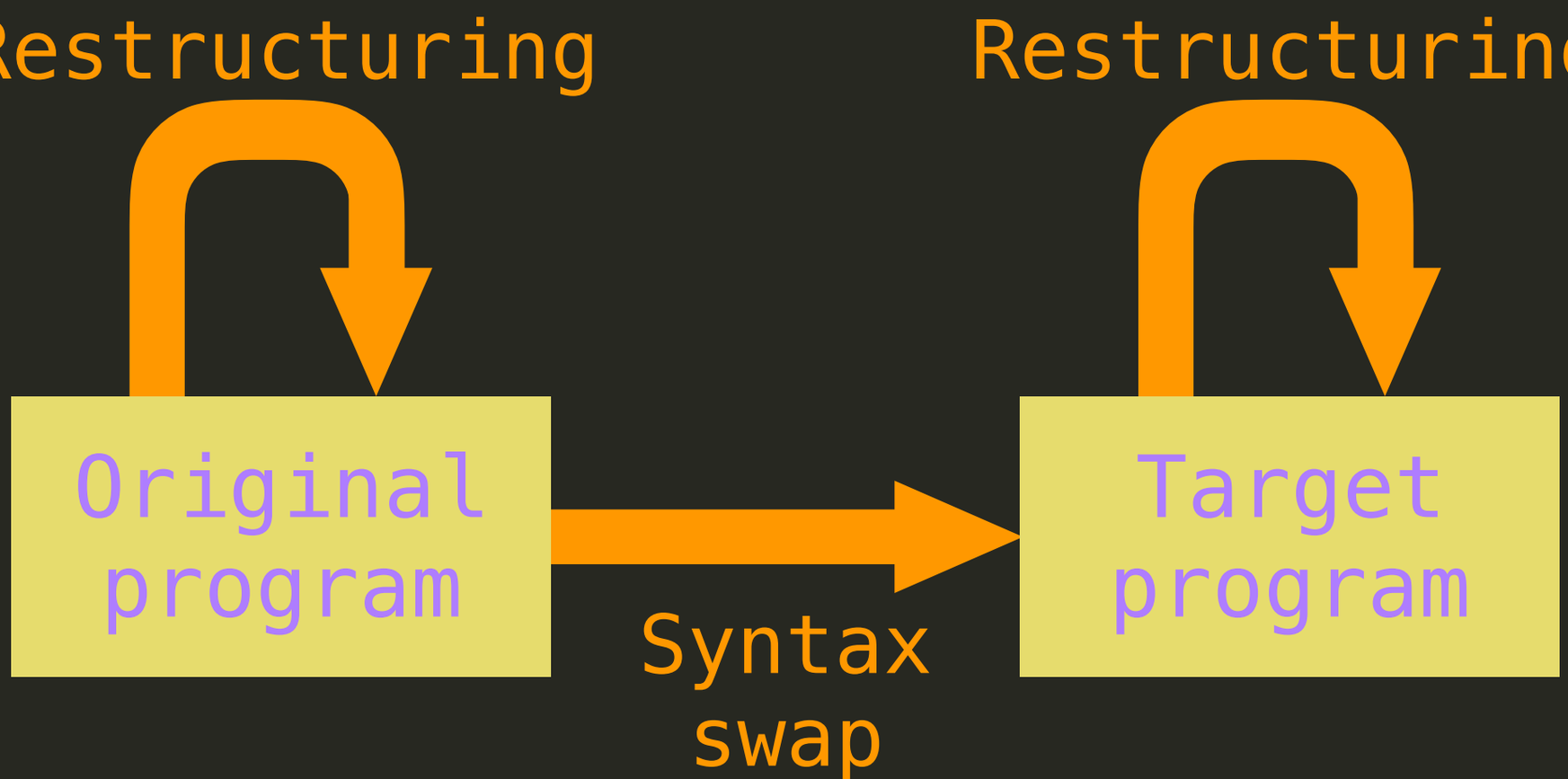
A.A.Terekhov, C.Verhoef, The Realities of Language Conversions, IEEE Software 2000.

Language Conversion



A.A.Terekhov, C.Verhoef, The Realities of Language Conversions, IEEE Software 2000.

Language Conversion



A.A.Terekhov, C.Verhoef, The Realities of Language Conversions, IEEE Software 2000.

Codegen properties

* Correctness
* Speed
* Size
* Memory use
* Network demands
* Energy
* . . .

F.Ferreira, B.Pientka, Bidirectional Elaboration of Dependently Typed Programs, PPDP 2014.

Correct codegen

* semantic preservation
  * …under special conditions
* protect from logical errors
  * verification
  * testing

Bit flip

* Software-Implemented Hardware Fault Tolerance (SIHFT)
* Measurement unit:
  * FIT (Failure in 1000000000 hours ≈114155 years)
* Reasons for SEU (Single Event Upsets)
  * natural radiation
  * chip temperature instability
  * malicious intervention
  * experimental technology
* Known victims
  * Sun, Toyota

M.Heing-Becker, T.Kamph, S.Schupp, Bit-error injection for software developers, CSMR-WCRE 2014

Fast code

* Optimisation
  * traditional semantic-preserving
* Supercompilation
  * partial evaluation
* Folding/unfolding
  * inlining functions

Code optimisation

* By basic blocks
* Construct data dependency graphs
* Convert to SSA
  * (Static Single Assignment)
* Eliminate common subexpressions
* Form a ladder sequence
* Allocate registers, pseudo-, memory…

D.Grune, K.v.Reeuwijk, H.E.Bal, C.J.H.Jacobs, K.Langendoen, Modern Compiler Design, 2ed, 2012, §9.1.2.

Code optimisation

* By rewriting
* Prepare instruction patterns
  * “load constant”, “multiply registers”, “add from memory”, etc
* Traverse the tree bottom-up thrice
  * Instruction collection
  * Instruction selecting
  * Code generation

D.Grune, K.v.Reeuwijk, H.E.Bal, C.J.H.Jacobs, K.Langendoen, Modern Compiler Design, 2ed, 2012, §9.1.4.

Folding/unfolding

* If code occurs several times
  * fold into a function and call it
* If a function is scarcely called
  * unfold its body
* Balancing
  * statically: with thresholds
  * dynamically: search-based

Folding/unfolding

* Function inlining
  * void f
    { ...

```
     print_square( i++ );
    ...
  }
void print_square(int n)
 { printf ("square = %d\n", n*n); }
```

D.Grune, K.v.Reeuwijk, H.E.Bal, C.J.H.Jacobs, K.Langendoen, Modern Compiler Design, 2ed, 2012, §7.3.3.

Folding/unfolding

* Function inlining
  * void f
    { ...

```
      printf ("square = %d\n", (i++)*(i++));
    ...
  }
void print_square(int n)
 { printf ("square = %d\n", n*n); }
```

D.Grune, K.v.Reeuwijk, H.E.Bal, C.J.H.Jacobs, K.Langendoen, Modern Compiler Design, 2ed, 2012, §7.3.3.

Folding/unfolding

* Function inlining
  * void f
    { ...
    {int n=i++;
     printf ("square = %d\n", n*n); }
    ...
  }

```
void print_square(int n)
 { printf ("square = %d\n", n*n); }
```

D.Grune, K.v.Reeuwijk, H.E.Bal, C.J.H.Jacobs, K.Langendoen, Modern Compiler Design, 2ed, 2012, §7.3.3.

Folding/unfolding

* Function inlining + supercompilation
  * void f
    { ...
    {int n=3;
     printf ("square = %d\n", n*n); }
    ...
  }

```
void print_square(int n)
 { printf ("square = %d\n", n*n); }
```

D.Grune, K.v.Reeuwijk, H.E.Bal, C.J.H.Jacobs, K.Langendoen, Modern Compiler Design, 2ed, 2012, §7.3.3.

Folding/unfolding

* Function inlining + supercompilation
  * void f
    { ...

```
      printf ("square = %d\n", 9);
    ...
  }
void print_square(int n)
 { printf ("square = %d\n", n*n); }
```

D.Grune, K.v.Reeuwijk, H.E.Bal, C.J.H.Jacobs, K.Langendoen, Modern Compiler Design, 2ed, 2012, §7.3.3.

Size matters

* Aggressive suppression of unused code
* Virtual machines + intermediate code
* Honest compression (e.g., LZ)
* Reliance on hardware
* Traditional optimisations

D.Grune, K.v.Reeuwijk, H.E.Bal, C.J.H.Jacobs, K.Langendoen, Modern Compiler Design, 2ed, 2012, §9.2

Superoptimisation

```
signum (x)
int x;
{
   if (x>0)       return  1;
   else if (x<0) return -1;
   else           return  0;
}
```

H.Massalin. Superoptimizer: A look at the smallest program, ASPLOS 1987.

Superoptimisation

```
(x in d0)
add.1  d0, d0
subx.1 d1, d1
negx.1 d0
addx.1 d1, d1
(result in d1)
```

H.Massalin. Superoptimizer: A look at the smallest program, ASPLOS 1987.

Superoptimisation

* State of the art:
  * brute force is viable
  * optimisation database caches solutions
    * enumerate everything possible
    * harvest beforehand
    * canonicalisation up to equivalence
    * stochastic search for large code segments
    * adapting memory subsystems
    * used in GCC (prooflink: PLDI-1992-GranlundK)

J.G.Wingbermuehle, R.K.Cytron, R.D.Chamberlain, Superoptimization of Memory Subsystems, LCTES 2014. http://bibtex.github.io/LCTES-2014-WingbermuehleCC.html

Power consumption

* More computation occurs on gadgets
* Save energy to increase optime
  * Reduce costs
  * Limit peak heat dissipation
  * Conceptually (weakly) linked to
    * sustainability

D.Grune, K.v.Reeuwijk, H.E.Bal, C.J.H.Jacobs, K.Langendoen, Modern Compiler Design, 2ed, 2012, §9.3

Power consumption

* Fast code uses less energy
  * `gcc -O1` saves 20%
* P = V × A
  * reduce CPU voltage
* Replace the scheduler
  * minimise changed bits

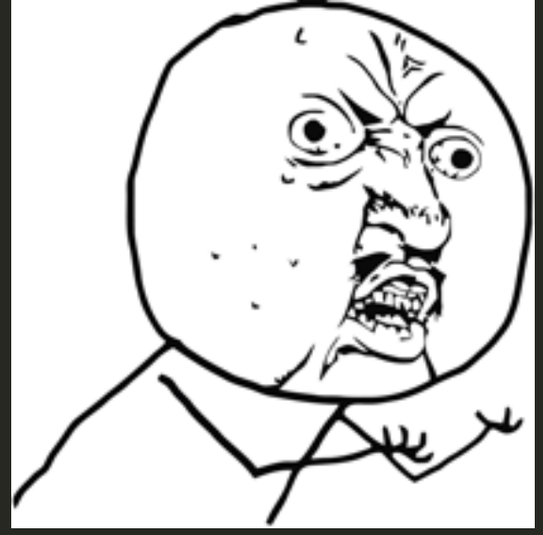
D.Grune, K.v.Reeuwijk, H.E.Bal, C.J.H.Jacobs, K.Langendoen, Modern Compiler Design, 2ed, 2012, §9.3

Part II Conclusion

* Language conversion does not exist
* Code should be correct
  * even with bit flips
* Code can be made small
  * Nothing beats superoptimisation
* Try to conserve energy
  * just make it fast and optimal

Conclusion

* General technology is easy
* Topics touched
  * Compilation: up, down, generative
  * Optimisation: speed, size, power
* Topics untouched
  * test suite optimisation
  * software transplantation



Y U NO SUBMIT REVIEW?!

Questions?