

Using Partial Evaluation to Generate Compilers

Partial Evaluation – Introduction

- Optimization technique (Program Specialization)
 - Evaluate programs with partial input
- Paradigm connected to program generation, program analysis, ...
- Tool for compiler generation

S_n^m - Theorem

- Given a function f with $m + n$ arguments it is possible to construct a function g with n arguments such that $f(s_1, \dots, s_m, d_1, \dots, d_n) = g(d_1, \dots, d_n)$
- Proven by S. Kleene (1943)
- Specialized functions could be slower than generic ones

Program Specialization

- Given a **program** f with $m + n$ arguments it is possible to construct a **program** g with n arguments such that $f(s_1, \dots, s_m, d_1, \dots, d_n) = g(d_1, \dots, d_n)$
- Specialized programs can be optimized
 - Validation of arguments is no longer required
 - Static computations are possible
 - Parts of control flow are known
 - ...

⇒ Partial Evaluators can optimize programs!

Program Specialization

Important: Program Specialization (Partial Evaluation) is not Partial Application or Currying

```
/* Generic method */
def pow(base: Int, exp: Int): Int =
  if (exp == 0) return 1
  else return base * pow(base, exp - 1)

/* Specialized method via partial application */
def pow4(base: Int): Int =
  pow(base, 4)

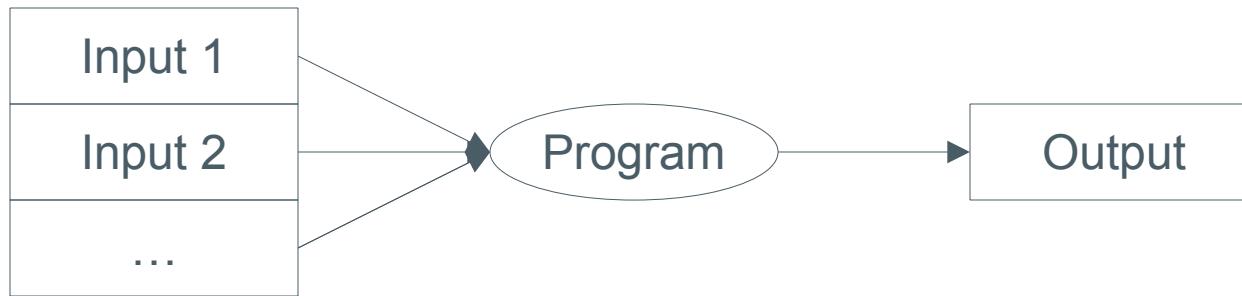
/* Specialized method via partial evaluation */
def pow4(base: Int): Int =
  return base * base * base * base
```

Execution & Programs

Executing Programs

- Execution in one go
 - All inputs have to be provided
 - One output is produced

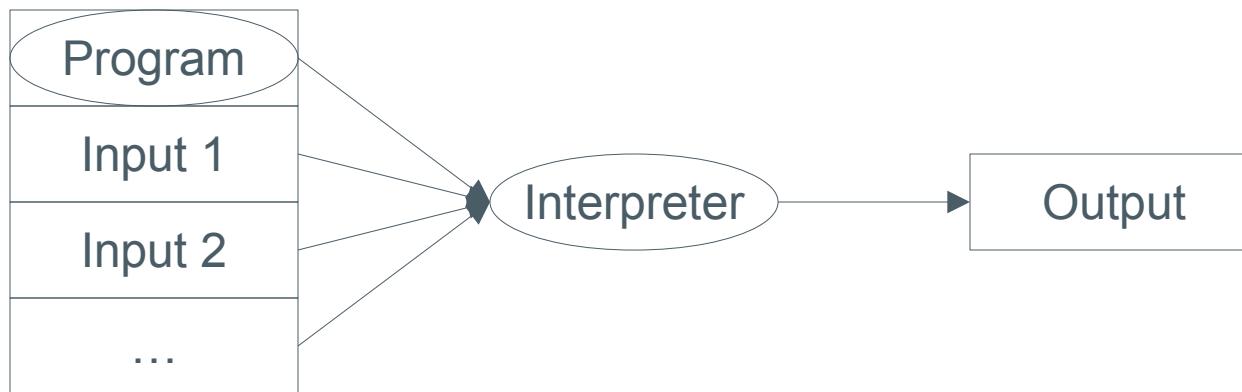
⇒ A single *computational stage*



Usually this is **not** how programs are executed!

Interpreter

- Meaning is assigned and directly executed
⇒ One computational stage
- Slow: Analysis and execution share resources
- Easy to implement



```
/* abstract syntax for expressions */
sealed trait Exp
case class Number(value: Int) extends Exp
case class Variable(name: String) extends Exp
case class Comp(lhs: Exp, op: (Int, Int) => Boolean, rhs: Exp) extends Exp
case class Arith(lhs: Exp, op: (Int, Int) => Int, rhs: Exp) extends Exp

/* abstract syntax for statements */
sealed trait Stm
case class If(condition: Compare, ifTrue: Stm, ifFalse: Stm) extends Stm
case class Assign(name: String, value: Exp) extends Stm
case class Seq(first: Stm, second: Stm) extends Stm
```

```
sealed trait Exp {  
    def eval(environment: Map[String, Int]): Int = this match {  
        case Number(value) =>  
            value  
  
        case Variable(name) =>  
            environment(name)  
  
        case Comp(lhs, op, rhs) =>  
            if (op(lhs.eval(environment), rhs.eval(environment))) 1 else 0  
  
        case Arith(lhs, op, rhs) =>  
            op(lhs.eval(environment), rhs.eval(environment))  
    }  
}
```

```
sealed trait Stm {
    def eval(environment: Map[String, Int]): Map[String, Int] = this match {
        case If(condition, ifTrue, iffFalse) =>
            if (condition.eval(environment) != 0) ifTrue.eval(environment)
            else iffFalse.eval(environment)

        case Assign(name, value) =>
            environment + (name -> value.eval(environment))

        case Seq(first, second) =>
            second.eval(first.eval(environment))
    }
}
```

```
val program = If(Comp(  
    Arith(Variable("x"), _ * _, Number(2)),  
    _ > _, Number(12)),  
    Assign("result", Number(42)),  
    Assign("result", Number(12)))  
  
program.eval(Map("x" -> 6))  
  
program.eval(Map("x" -> 7))
```

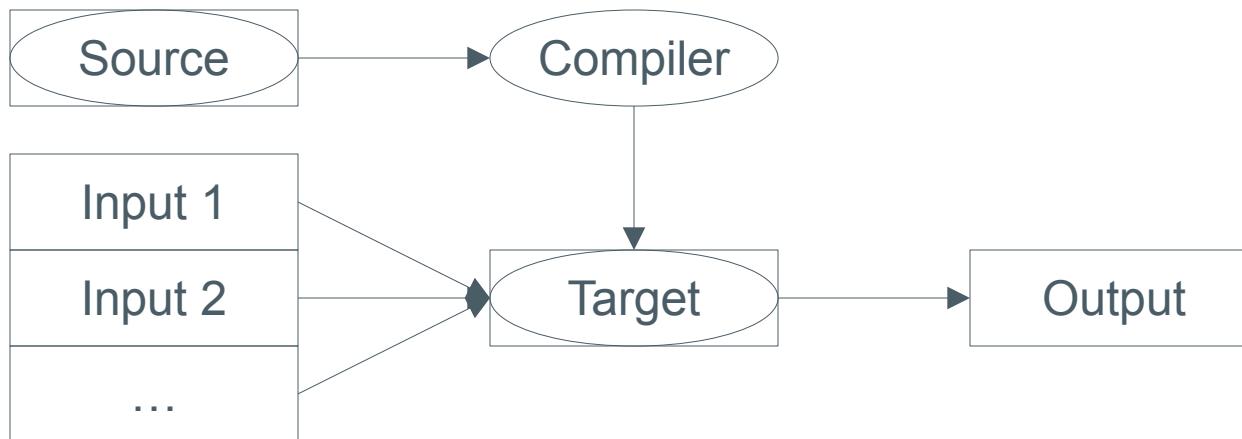
// if (
// (x * 2)
// > 12)
// then { result := 42; }
// else { result := 12; }

// x -> 6, result -> 12

// x -> 7, result -> 42

Compiler

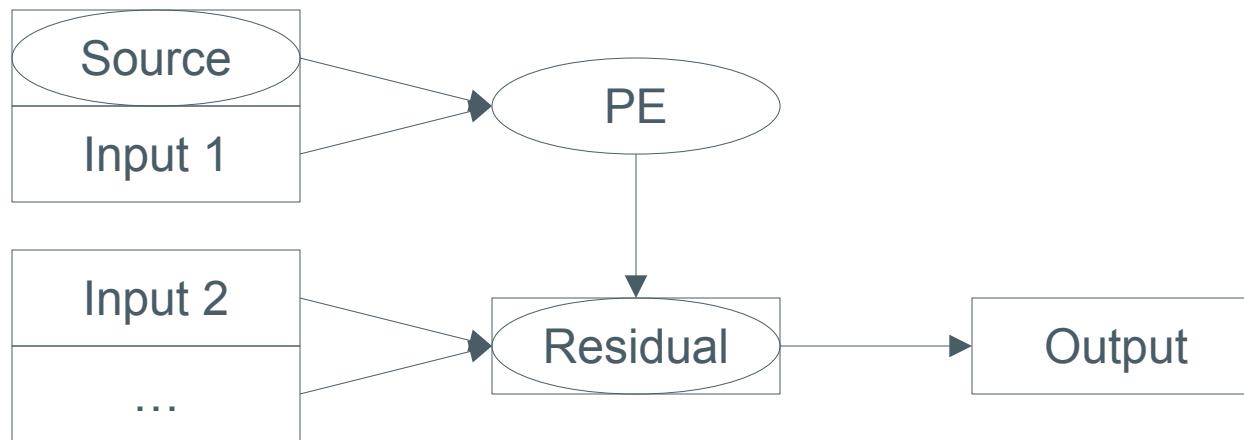
- Meaning is assigned and translated to another language
⇒ Two computational stages
- Faster: Analysis and execution are separated as different programs
- More complex implementation



Partial Evaluator

- Mix of interpretation (execute now) and compilation (execute later)
- Generalizes creation of computational stages

Operation: Given program and fixed inputs, specialized program is generated



How Do Partial Evaluators Work?

Program Specialization as an Optimization Strategy

Three problems have to be solved:

- 1) Execute static computations
- 2) Defer dynamic computations
- 3) Distinguish between static and dynamic computations

Defer Dynamic Computations

- Only parts of input are fixed/known
- Computations may depend on unknown input
- Partial Evaluator must generate a program containing these computations
 - ⇒ The **residual program** performs computations at runtime

Nothing really changes...

```
int example(int x, int y) {  
    int result = 1;  
    while (result < x) {  
        result += result * 3 / x;  
    }  
    return result;  
}
```

```
int example_2(int x /* int y = 2 */) {  
    int result = 1;  
    while (result < x) {  
        result += result * 3 / x;  
    }  
    return result;  
}
```

... Except (Optionally) the Output Language!

```
int example(int x, int y) {  
    int result = 1;  
    while (result < x) {  
        result += result * 3 / x;  
    }  
    return result;  
}
```

```
def example_2(x /* y = 2 */):  
    result = 1  
    while (result < x):  
        result += result * 3 // x  
  
    return result
```

Execute Static Computations

- Specialization as program optimization
- **What** can be optimized?
 - Expressions
 - Program Points
- **How** can we optimize?
 - Symbolic Computation
 - Unfolding of function calls
 - Program point specialization
 - ... many more!

Execute Static Computations – Symbolic Computation

- Computation depend on symbols and not (just) values
 - Arithmetic simplifications (simplification of expression structure)
 - Constant propagation
 - Constant folding
 - ...
- Reduces / Removes computations
- Can spread static values

Symbolic Computation – Example

```
proc p(x: int, y: int) {  
    printi(x);  
    if (y = 0) x := x + y;  
    printi(x * 2);  
}
```

```
proc p(/* x: int = 1 */ y: int) {  
    printi(1);  
    if (y = 0) x := x + y; // Propagate constant y  
    printi(x * 2);  
}
```

```
proc p(/* x: int = 1 */ y: int) {
    printi(1);
    if (y = 0) x := x + 0; // Remove statement and branch
    printi(x * 2);
}
```

```
proc p(/* x: int = 1 */ y: int) {  
    printi(1);  
  
    printi(x * 2); // Propagate and fold constant x  
}
```

```
proc p(/* x: int = 1 */ y: int) {  
    printi(1);  
  
    printi(2);  
}
```

Execute Static Computations – Unfolding of Calls

- Include body of (recursive) calls at call site
 - Known as *inlining* in most productive compilers today
- Allows further optimizations

Disadvantages:

- Increases code size
- Termination of optimization is not guaranteed

Unfolding of Calls – Example 1

```
int pow(int base, int exponent) {  
    if (exponent == 0) return 1;  
    else return (base * pow(base, exponent - 1));  
}
```

```
int pow_3(int base, /* int exponent = 3 */) {
    if (3 == 0) return 1; // Branch is not executed!
    else return (base * pow(base, 3 - 1)); // Unfold!
}
```

```
int pow_3(int base, /* int exponent = 3 */) {
    return base *
        if (2 == 0) return 1;
        else return (base * pow(base, 2));
} // Unfold again! And again! And ...
```

```
int pow_3(int base, /* int exponent = 3 */) {
    return base *
           base *
               base *
                   if (0 == 0) return 1; // Is 0 == 0? Yes!
                   else return (base * pow(base, 0 - 1));
}
```

```
int pow_3(int base, /* int exponent = 3 */) {
    // Arithmetic simplification!
    return base * base * base * 1;
}
```

```
int pow_3(int base, /* int exponent = 3 */) {
    return base * base * base;
}
```

Partial Evaluation works and is a powerful optimization strategy!

Unfolding of Calls – Example 2

```
def collatz: Int => Int = {
    case 1 => 0
    case n if n % 2 == 0 =>
        1 + collatz(n / 2)
    case n if n % 2 != 0 =>
        1 + collatz(3 * n + 1)
}
```

Unfolding of Calls – Example 2

```
def collatz_1 = 0
def collatz_2 = 1
def collatz_3 = 7
def collatz_4 = 2
...
...
```

Partial Evaluation works ... in these cases!

⇒ But we have to ***actively prevent*** infinite loops

Execute Static Computations – Program Point Specialization

- Creation of specialized definitions
- Parts of a program occur multiple times with identical specialized inputs
 - Only one specialized definition is required
 - Specialized definition can be shared between instances

Disadvantages:

- Potential increase in code size (not as much as unfolding)
- Termination of optimization is not guaranteed

Program Point Specialization – Example 1

```
def ack(m: Int, n: Int): Int =  
    if (m == 0) n + 1  
    else if (n == 0) ack(m - 1, 1)  
    else ack(m - 1, ack(m, n - 1))
```

```
def ack_2(/* m: Int = 2 */ n: Int): Int =  
    if (2 == 0) n + 1 // Is 2 == 0? No!  
    else if (n == 0) ack(1, 1)  
    else ack(1, ack(2, n - 1))  
  
// Multiple recursive calls with m = 2 and m = 1
```

```
def ack_2(/* m: Int = 2 */ n: Int): Int =
  else if (n == 0) ack_1(1)
  else ack_1(ack_2(n - 1))

def ack_1(/* m: Int = 1 */ n: Int): Int =
  if (1 == 0) n + 1
  else if (n == 0) ack(0, 1)
  else ack(0, ack(1, n - 1))
```

```
def ack_2(/* m: Int = 2 */ n: Int): Int =
  else if (n == 0) ack_1(1)
  else ack_1(ack_2(n - 1))

def ack_1(/* m: Int = 1 */ n: Int): Int =
  if (1 == 0) n + 1
  else if (n == 0) ack_0(1)
  else ack_0(ack_1(n - 1))

def ack_0(/* m: Int = 0 */ n: Int): Int =
  if (0 == 0) n + 1 // Is 0 == 0? Yes!
  else if (n == 0) ack(-1, 1)
  else ack(-1, ack(0, n - 1))
```

```
def ack_2(/* m: Int = 2 */ n: Int): Int =
  else if (n == 0) ack_1(1)
  else ack_1(ack_2(n - 1))

def ack_1(/* m: Int = 1 */ n: Int): Int =
  if (1 == 0) n + 1
  else if (n == 0) ack_0(1)
  else ack_0(ack_1(n - 1))

def ack_0(/* m: Int = 0 */ n: Int): Int =
  if (0 == 0) n + 1
  // ack_0 is not recursive! Unfold it!
```

```
def ack_2(/* m: Int = 2 */ n: Int): Int =
  else if (n == 0) ack_1(1)
  else ack_1(ack_2(n - 1))

def ack_1(/* m: Int = 1 */ n: Int): Int =
  if (1 == 0) n + 1
  else if (n == 0) 2
  else ack_1(n - 1) + 1
```

Distinguish between static and dynamic computations

- Static computations can be performed by the Partial Evaluator
- Dynamic computations have to be included in the residual program
- **Division** between static and dynamic depends on input
- Two approaches for division:
 - Offline Partial evaluation
 - Online Partial evaluation

Online Partial Evaluation

- Division is created “on the fly“ during specialization
- Fixed input is given (similar to environment in interpreter)
- Program is traversed, performing calculations and optimizations
 - Values are used to decide control flow, etc.
 - Usually yields better optimization

⇒ We performed *online* partial evaluation!

Offline Partial Evaluation

- Division is created during separate analysis
 - Binding Time Analysis
- Annotations in source code decide which parts are static and dynamic
 - Inserted by programmer
 - Propagated, analyzed automatically
- Conservative approach
 - Depends on knowledge of what inputs are static/dynamic annotated
 - Does not utilize value of inputs

Offline Partial Evaluation – Example

Annotations mark dynamic computations

```
def ack(m: Int, n: Int): Int =  
    if (m == 0) n + 1  
    else if (n == 0) ack(m - 1, 1)  
         else ack(m - 1, ack(m, n - 1))
```

- 1) Compute non-underlined constructs
- 2) Generate residual code for underlined constructs

```
def ack_2(/* m = 2 */ n: Int): Int =  
  if (n == 0) ack(1, 1)  
  else ack(1, ack(2, n - 1))
```

```
def ack_2(/* m = 2 */ n: Int): Int =
  if (n == 0) ack_1(1)
  else ack_1(ack_2(n - 1))

def ack_1(/* m = 1 */ n: Int): Int =
  if (n == 0) ack_0(1)
  else ack_0(ack_1(n - 1))

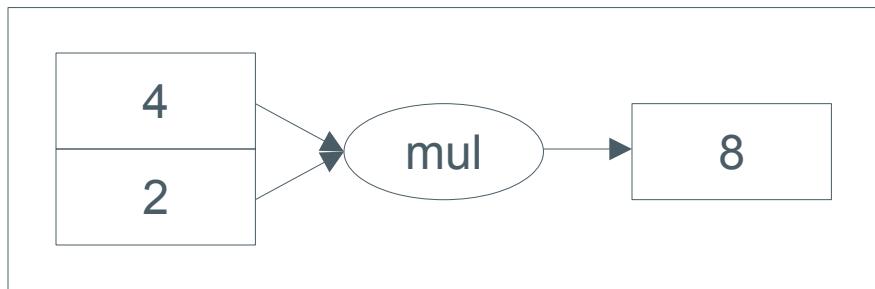
def ack_0(/* m = 0 */ n: Int): Int =
  n + 1
```

Offline partial evaluation often yields less optimized results

Multi-Stage Computation & Futamura Projections

Computations as Equations

- The execution of programs can be described with equations
 - Programs are functions
 - Inputs are arguments to this function
 - Outputs are result of function



$8 = \text{[mul]} (4, 2)$

Note: Equality describes the output of both sides, not the way it is produced!

```
8 = [mul] (4, 2)
    = [add] (5, 3)
```

```
9 = [collatz] (12)
    = [collatz] (13)
    = [collatz_12] ()
```

```
output = [example] (x, y)
        = [example] (x, 2)
        = [example_2] (x)
```

```
output = [[program]] (input1, input2)
= [[interpreter]] (program, input1, input2)
= [[compiler]] (program) ] (input1, input2)
```

```
output = [[program]] (input1, input2)
= [[ [pe] (program, input1) ] ] (input2)
= [ programinput1 ] (input2)
```

Computations as Equations – Partial Evaluators

- Partial Evaluators can separate „function arguments“
 - Programs with two inputs become two programs with one input
 - Additional computational stages are introduced
- Equations show computational stages as nested $\llbracket \rrbracket$ brackets
- Semantic equality of programs can be shown independent of comp. stages

Futamura Projections

- Yoshihiko Futamura presented projections in 1970s
- Partial Evaluation can not only optimize programs
 - Some combinations seemingly create new functionality
- In total: **3** Projections
- Links between *Partial Evaluation* and *Compiler Construction*

First Futamura Projection

```
output = [interpreter] (program, input)
```

- An interpreter accepts multiple inputs
- ⇒ Partial Evaluator can specialize an Interpreter
with respect to a source program!

First Futamura Projection

```
output = [interpreter] (program, input)
        = [[pe] (interpreter, program)] (input)
        =           [target]          (input)
```

- Specialization of an Interpreter with respect to a program yields:
Compiled Program
- Partial Evaluator can act as compiler!

```
val program = If(Comp(  
    Arith(Variable("x"), _ * _, Number(2)),           // if (  
    _ > _, Number(12)),                            // (x * 2)  
    Assign("result", Number(42)),                   // > 12 )  
    Assign("result", Number(12)))                    // then { result := 42; }  
                                                // else { result := 12; }  
  
program.eval (input) // Specialize interpreter to fixed program
```

```
if (Comp(
    Arith(Variable("x"), _ * _, Number(2)),
    _ > _, Number(12)).evaluate(input) != 0) // Unfold .evaluate
    Assign("result", Number(42)).evaluate(input)
else
    Assign("result", Number(12)).evaluate(input)
```

```
if (Arith(Variable("x"), _ * _, Number(2))
    .evaluate(input) >
    Number(12).evaluate(input)) // Unfold .evaluate again
    Assign("result", Number(42)).evaluate(input)
else
    Assign("result", Number(12)).evaluate(input)
```

```
if (Variable("x").evaluate(input) * Number(2).evaluate(input))  
    > 12) // Unfold .evaluate again ...  
    Assign("result", Number(42)).evaluate(input)  
else  
    Assign("result", Number(12)).evaluate(input)
```

```
if (input("x") * 2 > 12)
    Assign("result", Number(42)).evaluate(input) // Unfold .evaluate
else
    Assign("result", Number(12)).evaluate(input) // of assignments
```

```
if (input("x") * 2 > 12)           // if (x * 2 > 12)
    input + ("result" -> 42)      // then { result := 42; }
else
    input + ("result" -> 12)      // else { result := 12; }
```

Source program was translated to another language!

Second Futamura Projection

```
target = [pe] (interpreter, program)
```

- A Partial Evaluator accepts multiple inputs
- ⇒ Partial Evaluator can specialize a Partial Evaluator
with respect to an Interpreter!

Second Futamura Projection

```
target = [pe] (interpreter, program)
        = [[pe]] (pe, interpreter) ] (program)
        = [compiler]           (program)
```

- Specialization of a Partial Evaluator with respect to an Interpreter yields:
Compiler
- Partial Evaluator can act as compiler generator!

Third Futamura Projection

```
compiler = [pe] (pe, interpreter)
```

- A Partial Evaluator accepts multiple inputs *again*
 - ⇒ Partial Evaluator can specialize a Partial Evaluator with respect to a Partial Evaluator!

Third Futamura Projection

```
compiler = [[pe]] (pe, interpreter)
= [[[pe] (pe, pe)]] (interpreter)
= [[compiler-gen]] (interpreter)
```

- Specialization of a Partial Evaluator with respect to a Partial Evaluator yields:
Compiler Compiler or Compiler Generator
- Partial Evaluator can act as compiler generator *generator!*

Fourth Futamura Projection?

```
compiler-gen = [[pe]] (pe, pe)
```

- A Partial Evaluator accepts multiple inputs *again*²
 - ⇒ Partial Evaluator can specialize a Partial Evaluator with respect to a Partial Evaluator!

Fourth Futamura Projection?

```
compiler-gen = [[pe]] (pe, pe)
                = [[[pe] (pe, pe)]] (pe)
                = [[[ [pe] (pe, pe) ] (pe) ]] (pe)
                ...
                ...
```

- No new functionality arises from further specialization
 - Nothing can change, since all parts of the equation are `pe`
- Speed-Ups are possible!

Applications & Limits

Applications of Partial Evaluation

- Programs with a fixed (or rarely changing) configuration
 - Web Servers with configuration file
 - Rendering of a fixed scene
- Domain Specific Languages
 - Specialization to programs removes interpretation overhead
- General Software Development

Limits of Partial Evaluation – Termination

- Limited depth of optimizations (Unfolding, Symbolic Computations)
 - Unfolding has to terminate with call to (generic) definition
- Limited amount of created definitions (Program Point Specialization)
 - Amount of definitions has to be limited



Limits of Partial Evaluation

- Speed-Up is hard to predict
 - Binding Time Analysis makes predictions easier (at cost of performance)
 - More analysis requires more time ⇒ Less overall gain
- Difficult to use without knowledge or experience
- Cannot invent new optimizations

Conclusion

Conclusion

- Partial Evaluation is **not** a replacement for compilers / compiler writers
 - Humans are required to implement optimization techniques
- Applications are limited
 - No “one size fits all”-solution
 - Varying benefits in different areas
- More development required (theoretical and practical)

Further Reading

- S. Kleene. “General Recursive Functions of Natural Numbers”. In: *Mathematische Annalen* 112.1 (Dec. 1936), pp. 727–742.
- N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. ISBN: 0130202495.
- N. Jones. “An Introduction to Partial Evaluation”. In: *ACM Computing Surveys* 28.3 (1996), pp. 480–503.
- Y. Futamura. “Partial Evaluation of Computation Process—An Approach to a CompilerCompiler”. In: *Higher-Order and Symbolic Computation* 12.4 (1999), pp. 381–391.

Discussion