Papers We Love Milano #4

Partial Computation of Programs

(Futamura 1983)

Edoardo Vacchi

19th February 2020

Allocation Removal by Partial Evaluation in a Tracing JIT

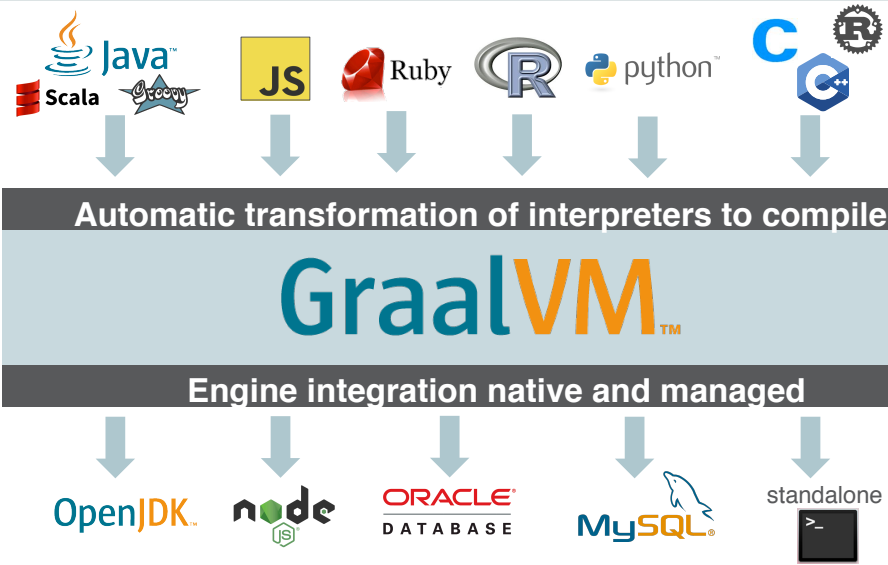
Carl Friedrich Bolz[a] Antonio Cuni[a] Maciej Fijałkowski[b] Michael Leuschel[a]
Samuele Pedroni[c] Armin Rigo[a]

[a]Heinrich-Heine-Universität Düsseldorf, STUPS Group, Germany
[b]merlinux GmbH, Hildesheim, Germany
[c]Open End, Göteborg, Sweden

cfbolz@gmx.de anto.cuni@gmail.com fijal@merlinux.eu leuschel@cs.uni-duesseldorf.de
samuele.pedroni@gmail.com arigo@tunes.org

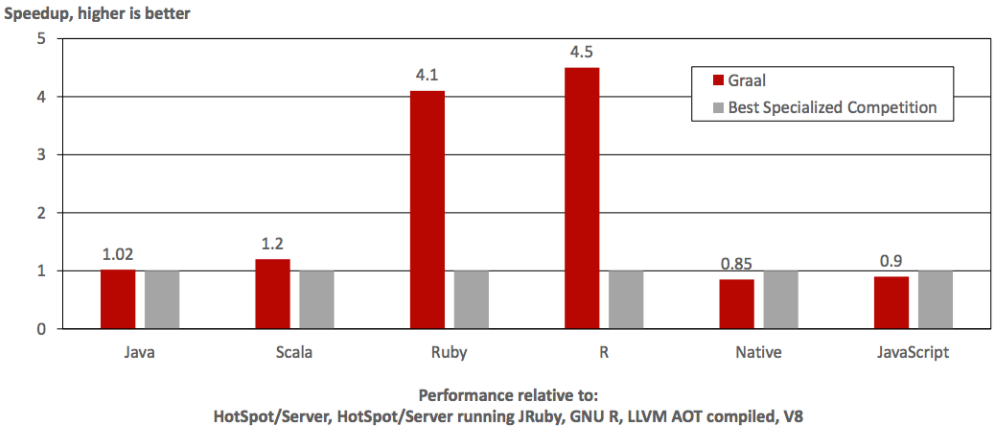
The performance of many dynamic language implementations suffers from high allocation rates and runtime type checks. This makes dynamic languages less applicable to purely algorithmic problems, despite their growing popularity. In this paper we present a simple compiler optimization based on online partial evaluation to remove object allocations and runtime type checks in the context of a tracing JIT. We evaluate the optimization using a Python VM and find that it gives good results for all our (real-life) benchmarks.

The performance of many dynamic language implementations suffers from high allocation rates and runtime type checks. This makes dynamic languages less applicable to purely algorithmic problems, despite their growing popularity. In this paper we present a simple compiler optimization based on online partial evaluation to remove object allocations and runtime type checks in the context of a tracing JIT. We evaluate the optimization using a Python VM and find that it gives good results for all our (real-life) benchmarks.



https://gotober.com/2018/sessions/650/graalvm-run-programs-faster-anywhere

Performance: Graal VM



https://gotober.com/2018/sessions/650/graalvm-run-programs-faster-anywhere

Practical Partial Evaluation for High-Performance Dynamic Language Runtimes

Thomas Würthinger[*] Christian Wimmer[*] Christian Humer[*] Andreas Wöß[*]
Lukas Stadler[*] Chris Seaton[*] Gilles Duboscq[*] Doug Simon[*] Matthias Grimmer[†]

[*]Oracle Labs [†]Institute for System Software, Johannes Kepler University Linz, Austria
{thomas.wuerthinger, christian.wimmer, christian.humer, andreas.woess, lukas.stadler, chris.seaton, gilles.m.duboscq, doug.simon}@oracle.com matthias.grimmer@jku.at

Most high-performance dynamic language virtual machines duplicate language semantics in the interpreter, compiler, and runtime system. This violates the principle to not repeat yourself. In contrast, we define languages solely by writing an interpreter. The interpreter performs specializations, e.g., augments the interpreted program with type information and profiling information. Compiled code is derived automatically using partial evaluation while incorporating these specializations. This makes partial evaluation practical in the context of dynamic languages: it reduces the size of the compiled code while still compiling all parts of an operation that are relevant for a particular program. When a speculation fails, execution transfers back to the interpreter, the program re-specializes in the interpreter, and later partial evaluation again transforms the new state of the interpreter to compiled code.

Most high-performance dynamic language virtual machines duplicate language semantics in the interpreter, compiler, and runtime system. This violates the principle to not repeat yourself. In contrast, we define languages solely by writing an interpreter. The interpreter performs specializations, e.g., augments the interpreted program with type information and profiling information. Compiled code is derived automatically using partial evaluation while incorporating these specializations. This makes partial evaluation practical in the context of dynamic languages: it reduces the size of the compiled code while still compiling all parts of an operation that are relevant for a particular program. When a speculation fails, execution transfers back to the interpreter, the program re-specializes in the interpreter, and later partial evaluation again transforms the new state of the interpreter to compiled code.

We implement the language semantics only once in a simple form: as a language interpreter written in a managed high-level host language. Optimized compiled code is derived from the interpreter using partial evaluation. This approach and its obvious benefits were described in 1971 by Y. Futamura, and is known as the *first Futamura projection*. To the best of our knowledge no prior high-performance language implementation used this approach.

We implement the language semantics only once in a simple form: as a language interpreter written in a managed high-level host language. Optimized compiled code is derived from the interpreter using partial evaluation. This approach and its obvious benefits were described in 1971 by Y. Futamura, and is known as the *first Futamura projection*. To the best of our knowledge no prior high-performance language implementation used this approach.

first Futamura projection


Futamura
PROJECTIONS
1983

Partial Computation of Programs

Yoshihiko Futamura

Central Research Laboratory, HITACHI,LTD.
Kokubunji, Tokyo, Japan

Abstract

This paper attempts to clarify the difference between partial and ordinary computation. Partial computation of a computer program is by definition "specializing a general program based upon its operating environment into a more efficient program". It also shows the usefulness of partial computation. Finally, the formal theory of partial computation, technical problems in making it practical, and its future research problems are discussed.

The main purpose of this paper is to make partial computation effectiveness widely known. However, two new results are also reported:

(1) a partial computation compiler, and

(2) a tabulation technique to terminate partial computation.

Partial Computation of Programs

Yoshihiko Futamura

Central Research Laboratory, HITACHI,LTD.
Kokubunji, Tokyo, Japan

Abstract

This paper attempts to clarify the difference between partial and ordinary computation. Partial computation of a computer program is by definition "specializing a general program based upon its operating environment into a more efficient program". It also shows the usefulness of partial computation. Finally, the formal theory of partial computation, technical problems in making it practical, and its future research problems are discussed.

The main purpose of this paper is to make partial computation effectiveness widely known. However, two new results are also reported:

(1) a partial computation compiler, and

(2) a tabulation technique to terminate partial computation.

6. Conclusion

In the last decade, the author never believed that a practical projection machine could be implemented within 10 years. However, he is now more optimistic.

The achievements of Ershov, Emanuelson, Kahn and others establish the basis for the theory and practice of partial computation. Furthermore, the comming of a commercial LISP machine and inauguration of the 5th generation computer project in Japan will encourage research in this field.

6. Conclusion

In the last decade, the author never believed that a practical projection machine could be implemented within 10 years. However, he is now more optimistic.

The achievements of Ershov, Emanuelson, Kahn and others establish the basis for the theory and practice of partial computation. Furthermore, the comming of a commercial LISP machine and inauguration of the 5th generation computer project in Japan will encourage research in this field.

Programs and Programming Languages

Programs

- We call a *program* a *sequence of instructions* that can be *executed* by a *machine*.
- The *machine* may be a *virtual* machine or a *physical* machine
- In the following, when we say that a *program* is *evaluated*, we assume that there exists some *machine* that is able to execute these instructions.

Computational Models

- "A sort of programming language"
- *Mechanical* evaluation

- Turing machines
- Partial recursive functions
- Church's Lambda expressions

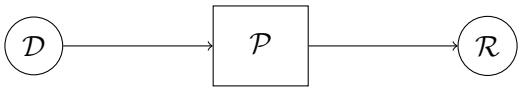
Computational Models

1. Conditional
2. Read/Write Memory
3. Jump (loop)

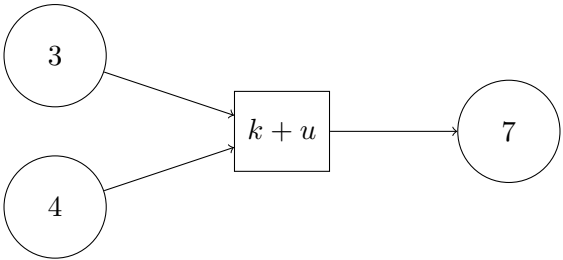
1. Condition
2. Expression
3. Function Definition

Program Evaluation

- Consider a program $\mathcal{P}$, with input data $\mathcal{D}$;
- when we *evaluate* $\mathcal{P}$ over $\mathcal{D}$ it produces some output result $\mathcal{R}$.

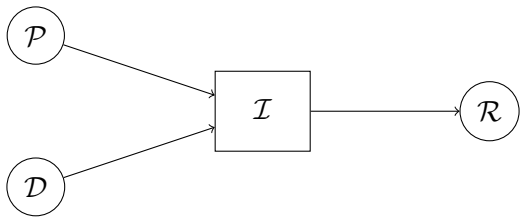


$$f(k, u) = k + u$$



Interpreters

- An *interpreter* $\mathcal{I}$ is a *program*
- it *evaluates* some other given program $\mathcal{P}$ over some given data $\mathcal{D}$, and it produces the output result $\mathcal{R}$.



- We denote this with $\mathcal{I}(\mathcal{P}, \mathcal{D})$

$f(k, u) = k + u$

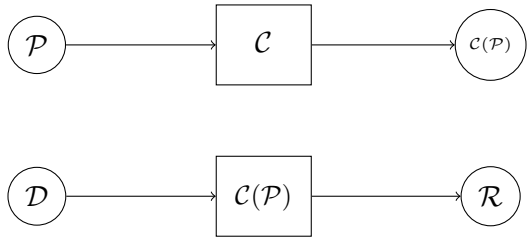
Instructions

```
add x y
sub x y
mul x y
...
```

write($\mathcal{D}$)
while(*has-more-instructions*($\mathcal{P}$)):
  instr $\leftarrow$ *fetch-next-instruction*($\mathcal{P}$)
  switch(*op*(instr)):
    case `'add'`:
      `x` $\leftarrow$ **read**()
      `y` $\leftarrow$ **read**()
      result $\leftarrow x + y$
      write(result)
    case ...

Compilers

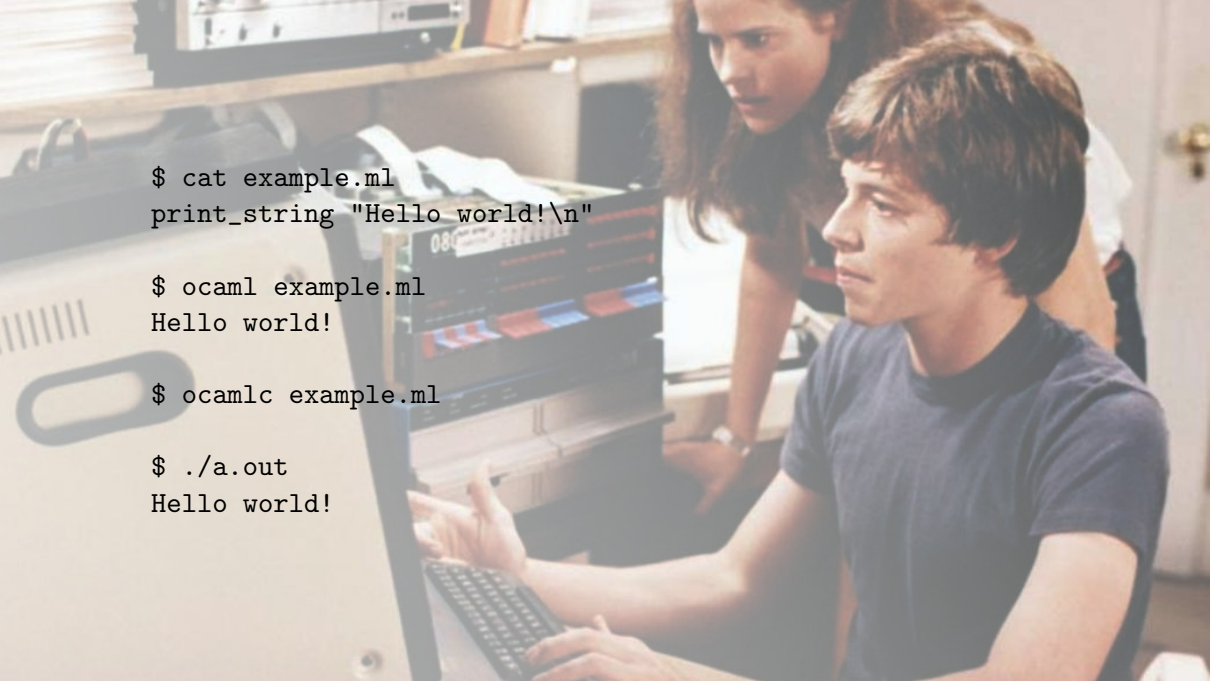
- Let be $\mathcal{P}$ a program that evaluates to $\mathcal{R}$ when given $\mathcal{D}$;
- A *compiler* $\mathcal{C}$ translates a **source program** $\mathcal{P}$ into an **object program** $\mathcal{C}(\mathcal{P})$ that evaluated over an input $\mathcal{D}$ still produces $\mathcal{R}$



- We denote this with $\mathcal{C}(\mathcal{P})(\mathcal{D})$

$f(k, u) = k + u$

```
sum:
    lea     eax, [rdi+rsi]
    ret
```



```
$ cat example.ml
print_string "Hello world!\n"

$ ocaml example.ml
Hello world!

$ ocamlc example.ml

$ ./a.out
Hello world!
```

$$\mathcal{C}(\mathcal{P})(\mathcal{D}) = \mathcal{I}(\mathcal{P}, \mathcal{D})$$

Partial Evaluation

Partial Evaluation (intuition)

Let us have a computation f of two parameters k, u

$$f(k, u)$$

- Now suppose that f is often called with $k = 5$;
- $f_5(u) :=$ "*f by substituting* 5 *for* k *and doing all possible computation based upon value* 5"
- Partial evaluation is *the process of transforming* $f(5, u)$ *into* $f_5(u)$

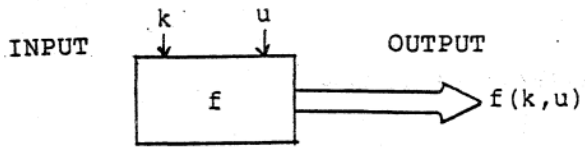


Fig 1: Total computation produces a final result.

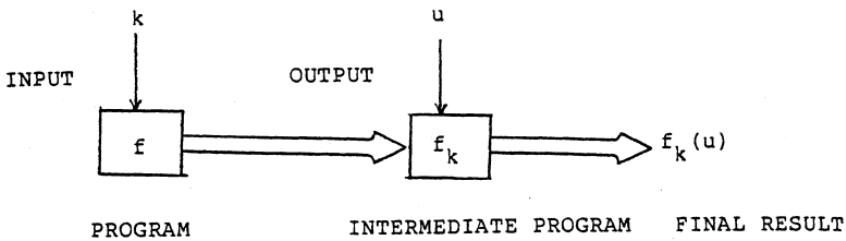


Fig 2: Partial computation produces intermediate program f_k based upon f and partial data k.

This is Currying! I Know This!

- Not exactly! In *functional programming currying* or *partial application*[a] is $f_5(u) := f(5, u)$

```
let f =
  (k, u) =>
    k * (k * (k+1) + u+1) + u*u;
let f5 = (u) => f(5, u);
```

- In a functional programming language this usually *does not change* the program that implements f

[a] Although, strictly speaking they are not synonyms, see https://en.wikipedia.org/wiki/Currying

Simplification

```
let f = (k, u) => k * (k * (k+1) + u + 1) + u * u;
```

by fixing $k = 5$ and *simplifying*:

```
let f5 = (u) => 5 * (31 + u) + u * u;
```

Rewriting

```
function pow(n, k) {
    if (k <= 0) {
        return 1;
    } else {
        return n * pow(n, k-1);
    }
}
function pow5(n) {
    return pow(n, 5);
}
```

Rewriting

```
function pow(n, k) {
    if (k <= 0) {
        return 1;
    } else {
        return n * pow(n, k-1);
    }
}
function pow5(n) {
    return n * pow(n, 4);
}
```

Rewriting

```
function pow(n, k) {
    if (k <= 0) {
        return 1;
    } else {
        return n * pow(n, k-1);
    }
}
function pow5(n) {
    return n * n * pow(n, 3);
}
```

Rewriting

```
function pow(n, k) {
    if (k <= 0) {
        return 1;
    } else {
        return n * pow(n, k-1);
    }
}
function pow5(n) {
    return n * n * n * n * n;
}
```

Rewriting

```
function pow(n, k) {
    if (k <= 0) {
        return 1;
    } else {
        return n * pow(n, k-1);
    }
}
function pow5(n) {
    return n * n * n * n * n;
}
```

In compilers this is sometimes called *inlining*

Rewriting and Simplification

- **Rewriting** is similar to macro expansion and *procedure integration* (*β-reduction*, *inlining*) in the optimization technique of a compiler.
- Often combined with **simplification** (*constant folding*)

ORIGINAL PL/I PROCEDURES

```
P:PROC(A);
  B=5;
  C=A*B;
  CALL Q(A,B);
RETURN(C);
END;

Q:PROC(X,Y);
  X=X*Y;
END;
```

AFTER INTEGRATION

```
P:PROC(A);
  B=5;
  C=A*B;
  A=A*B;
  RETURN(C);
END;
```

AFTER SIMPLIFICATION

```
P:PROC(A);
  A=A*5;
  RETURN(A);
END;
```

Projection

The following equation holds for f_k and f

$$f_k(u) = f(k, u) \tag{1}$$

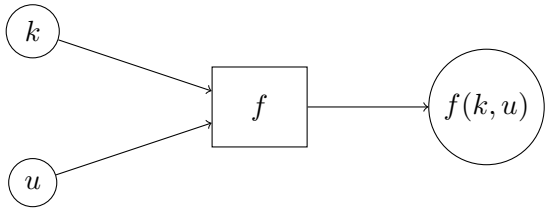
we call f_k **a projection of** f **at** k

Partial Evaluator

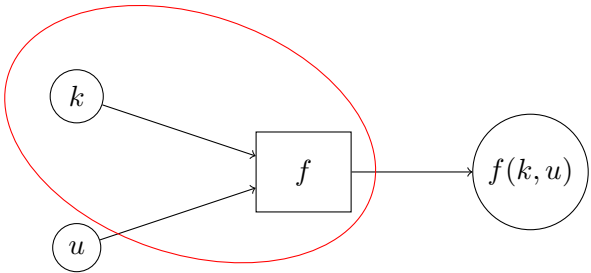
A partial computation procedure may be a computer program α called **a projection machine**, **partial computer** or **partial evaluator**.

$$\alpha(f, k) = f_k \tag{2}$$

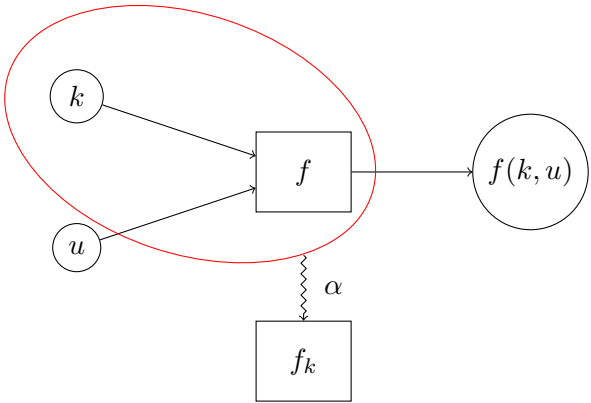
Partial Evaluator



Partial Evaluator



Partial Evaluator



Partial Evaluator

```
function pow(n, k) {
    if (k <= 0) {
        return 1;
    } else {
        return n * pow(n, k-1);
    }
}
let pow5 = alpha(pow, {k:5});
// (n) => n * n * n * n * n;
```

Examples

The paper presents:

- Automatic theorem proving
- Pattern matching
- Syntax analyzer
- Automatically generating a compiler

Examples

The paper presents:

- Automatic theorem proving
- Pattern matching
- Syntax analyzer
- Automatically generating a compiler

Interpreters and Compilers (*reprise*)

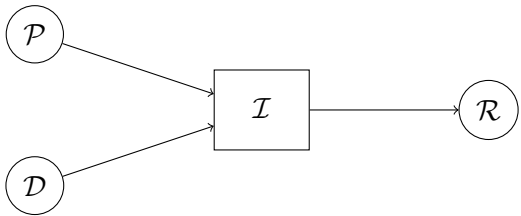
- An interpreter is a *program*
- This *program* takes another *program* and the data as input
- It evaluates the program on the input and returns the result

$$\mathcal{I}(\mathcal{P}, \mathcal{D})$$

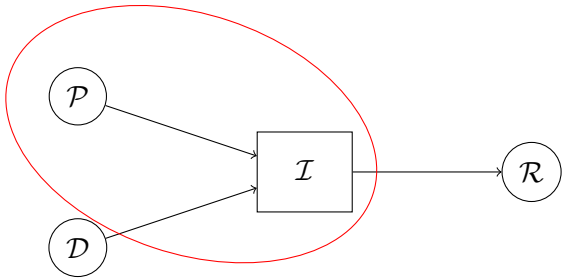
- A compiler is a *program*
- This *program* takes a **source program** and returns an *object program*
- The **object program** processes the input and returns the result

$$\mathcal{C}(\mathcal{P})(\mathcal{D})$$

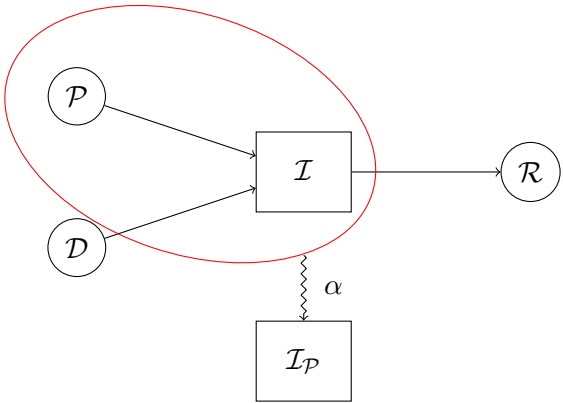
Partial Evaluation of an Interpreter



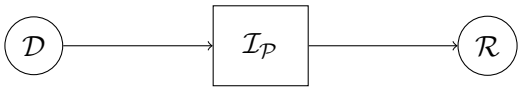
Partial Evaluation of an Interpreter



Partial Evaluation of an Interpreter



First Equation of Partial Computation (First Projection)



- That is, by feeding $\mathcal{D}$ into $\mathcal{I}_{\mathcal{P}}$, you get $\mathcal{R}$;
- in other words, $\mathcal{I}_{\mathcal{P}}$ is **an object program**.

$$\begin{aligned} \mathcal{I}(\mathcal{P},\mathcal{D}) &= \mathcal{C}(\mathcal{P})(\mathcal{D}) \\ \alpha(\mathcal{I},\mathcal{P}) &= \mathcal{I}_{\mathcal{P}} \\ \mathcal{I}_{\mathcal{P}} &= \mathcal{C}(\mathcal{P}) \end{aligned} \tag{4}$$

$f(k, u) = k + u$ `(add x y)`

write($\mathcal{D}$)
while(*has-more-instructions*($\mathcal{P}$)):
  instr $\leftarrow$ *fetch-next*($\mathcal{P}$)
  switch(*op*(instr)):
    case `'add'`:
      `x` $\leftarrow$ **read**()
      `y` $\leftarrow$ **read**()
      result $\leftarrow x + y$
      write(result)
    case ...

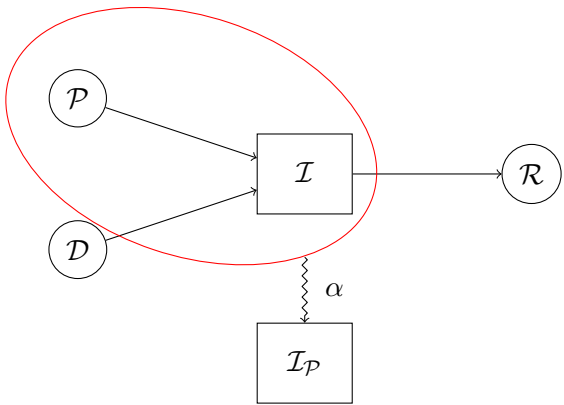
$f(k, u) = k + u$ `(add x y)`

write($\mathcal{D}$)
while(*has-more-instructions*($\mathcal{P}$)):
  instr $\leftarrow$ *fetch-next*($\mathcal{P}$)
  switch(*op*(instr)):
    case `'add'`:
      `x` $\leftarrow$ **read**()
      `y` $\leftarrow$ **read**()
      result $\leftarrow x + y$
      write(result)
    case ...

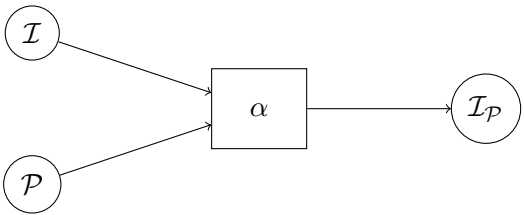
...but this interpreter executes on a *machine*!

```
sum:
    lea     eax, [rdi+rsi]
    ret
```

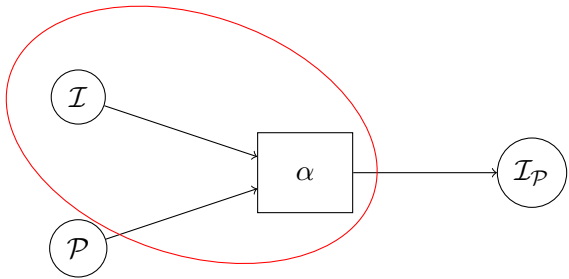
Partial Evaluation of an Interpreter



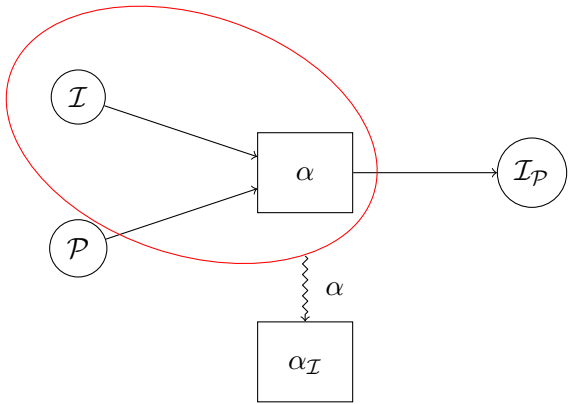
Partial Evaluation of an Interpreter



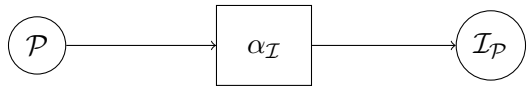
Partial Evaluation of the Partial Evaluation of an Interpreter



Partial Evaluation of an Interpreter



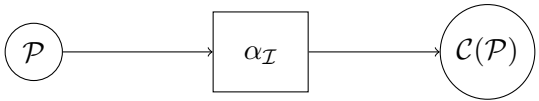
Second Equation of Partial Computation (Second Projection)



$$\alpha_{\mathcal{I}}(\mathcal{P}) = \mathcal{I}_{\mathcal{P}} \tag{5}$$

- but $\mathcal{I}_{\mathcal{P}}$, evaluated on $\mathcal{D}$ gives $\mathcal{R}$

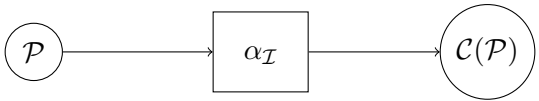
Second Equation of Partial Computation (Second Projection)



$$\alpha_{\mathcal{I}}(\mathcal{P}) = \mathcal{I}_{\mathcal{P}} \tag{5}$$

- but $\mathcal{I}_{\mathcal{P}}$, evaluated on $\mathcal{D}$ gives $\mathcal{R}$
- then $\mathcal{I}_{\mathcal{P}}$ is an object program ($\mathcal{P}' = \mathcal{C}(\mathcal{P})$)

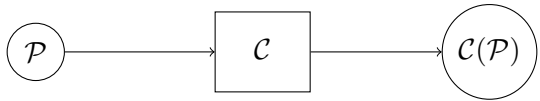
Second Equation of Partial Computation (Second Projection)



$$\alpha_{\mathcal{I}}(\mathcal{P}) = \mathcal{I}_{\mathcal{P}} \tag{5}$$

- but $\mathcal{I}_{\mathcal{P}}$, evaluated on $\mathcal{D}$ gives $\mathcal{R}$
- then $\mathcal{I}_{\mathcal{P}}$ is an object program ($\mathcal{P}' = \mathcal{C}(\mathcal{P})$)
- $\alpha_{\mathcal{I}}$ transforms a source program $\mathcal{P}$ to $\mathcal{I}_{\mathcal{P}}$ (*i.e.*, $\mathcal{C}(\mathcal{P})$)

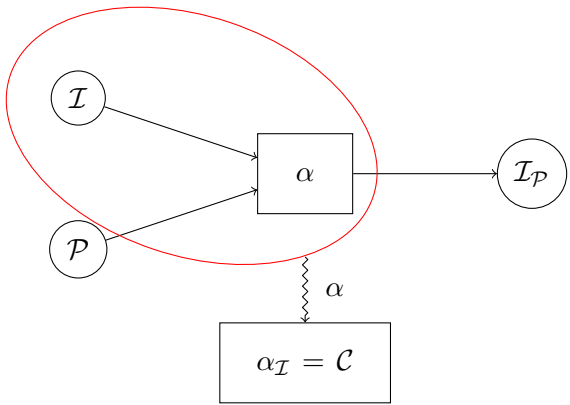
Second Equation of Partial Computation (Second Projection)



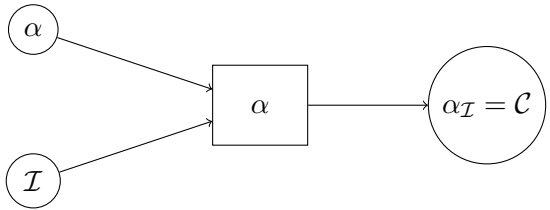
$$\alpha_{\mathcal{I}}(\mathcal{P}) = \mathcal{I}_{\mathcal{P}} \tag{5}$$

- but $\mathcal{I}_{\mathcal{P}}$, evaluated on $\mathcal{D}$ gives $\mathcal{R}$
- then $\mathcal{I}_{\mathcal{P}}$ is an object program ($\mathcal{P}' = \mathcal{C}(\mathcal{P})$)
- $\alpha_{\mathcal{I}}$ transforms a source program $\mathcal{P}$ to $\mathcal{I}_{\mathcal{P}}$ (*i.e.*, $\mathcal{C}(\mathcal{P})$)
- then $\alpha_{\mathcal{I}}$ is *a compiler*

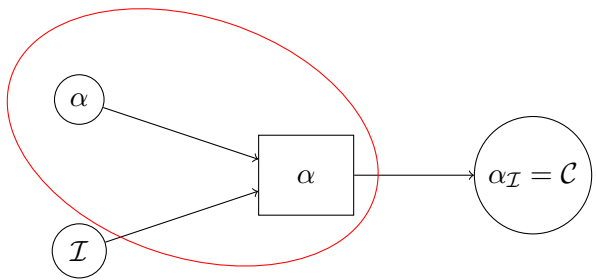
Partial Evaluation of the Partial Evaluation of an Interpreter



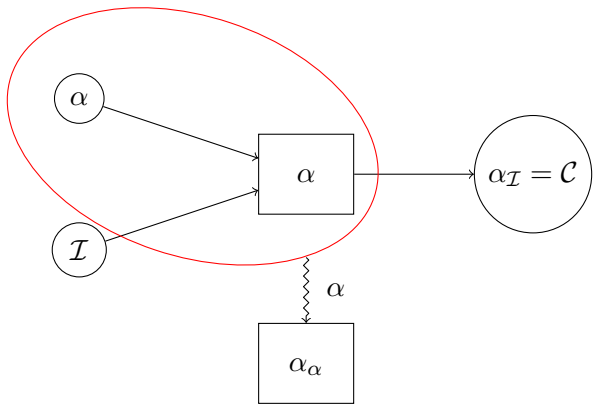
Partial Evaluation of the Partial Evaluation of an Interpreter



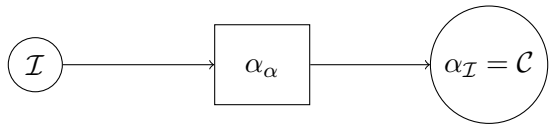
Partial Evaluation of the Partial Evaluation of an Interpreter



Partial Evaluation of the Partial Evaluation of the Partial Evaluation of an Interpreter



Third Equation of Partial Computation (Third Projection)



$$\alpha_\alpha(\mathcal{I}) = \alpha_\mathcal{I} \tag{6}$$

- α_α is a program that given $\mathcal{I}$, returns $\alpha_\mathcal{I} = \mathcal{C}$
- $\alpha_\mathcal{I}$ transforms a source program to an object program
- $\alpha_\mathcal{I}$ is a compiler
- α_α is a *compiler-compiler* (a *compiler generator*) which generates a compiler $\alpha_\mathcal{I}$ from an interpreter $\mathcal{I}$


MCE
XII-52

Partial Evaluation of a Partially-Evaluated Evaluator

- Let us call $\mathcal{I}$**-language** a language implemented by interpreter $\mathcal{I}$,

$$\alpha_\alpha(\mathcal{I}) = \alpha_\mathcal{I}$$

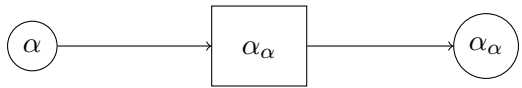
- $\alpha_\mathcal{I}$ is then a $\mathcal{I}$**-language compiler**
- let us now substitute α for $\mathcal{I}$ in $\alpha_\alpha(\mathcal{I}) = \alpha_\mathcal{I}$,
- which means considering α an interpreter for the α-language.

$$\alpha_\alpha(\alpha) = \alpha_\alpha$$

- α_α is an α**-language** compiler.

Fourth Equation of Partial Computation

$$\alpha_\alpha(\alpha) = \alpha_\alpha$$



- α_α is an α-language compiler.
- $\alpha_\alpha(\mathcal{I}) = \alpha_\mathcal{I}$ is an object program of $\mathcal{I}$; thus:

$$\alpha_\alpha(\mathcal{I})(\mathcal{P}) = \mathcal{I}_\mathcal{P} \tag{7}$$

- What is the α-language?

What is the α-language?

$$\begin{aligned} \alpha_\alpha(\mathcal{I})(\mathcal{P}) &= \mathcal{I}_\mathcal{P} \\ \alpha_\alpha(f)(k) &= f_k \end{aligned}$$

- In other words, by finding α_α we can generate the **partial computation** of f at k, f_k
- That is, α_α is a **partial evaluation compiler** (or *generator*).
- However, *the author notes*, at the time of writing, **there is no way to produce** α_α from $\alpha(\alpha, \alpha)$ for practical α's.

Theory of Partial Computation
and
Technical Problems

Conditions for a Projection Machines

1. **Correctness.** Program α must satisfy $\alpha(f, k)(u) = f(k, u)$
2. **Efficiency Improvement.** Program α should perform as much computation as possible for the given data k
3. **Termination.** Program α should terminate on partial computation of as many programs as possible. Termination at $\alpha(\alpha, \alpha)$ is most desirable

However, author notes, (2) is not *mathematically clear*

Computation Rule for Recursive Program Schema

Total Computation:

1. Rewriting
2. Simplification

Computation Rule for Recursive Program Schema

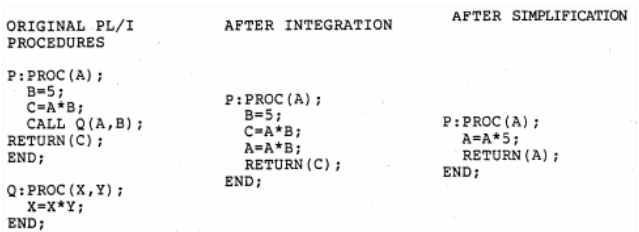
Partial Computation of f at k:

1. Rewriting (when semi-bound; e.g. $f(5, u)$)
2. Simplification
3. Tabulation

The discriminating character for p.c. are the *semi-bound call* and *tabulation*.

Rewriting and Simplification

Rewriting is similar to macro expansion and *procedure integration* in the optimization technique of a compiler. Often combined with **simplification.**



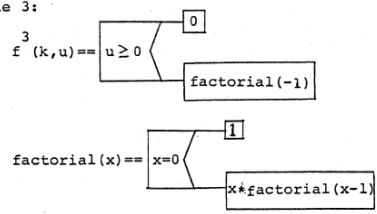
Tabulation is a process used to associate program f and known data k to projection f_k. Essentially, it builds a table, like that shown below, and searches through it.

function	known data	projection
f	k1	f_{k1}
f	k2	f_{k2}
g	k3	g_{k3}
g	k4	g_{k4}
f	k5	f_{k5}
.	.	.
.	.	.
.	.	.

For example, when there is a semi-bound call f(k2,u+1), it is replaced by $f_{k2}(u+1)$ and the table is searched for the entry of f, k2 and f_{k2}.

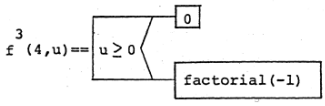
If the entry is found, partial computation of f(k2,u+1) is not performed again. If the entry is not found, f, k2 and the unfinished definition of f_{k2} are entered into the table and partial computation of f(k2,u+1) is performed. (A more detailed discussion of this is conducted in Chapter 4).

Example 3:



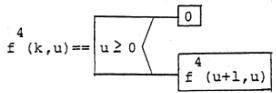
Example 3: Partially compute $f^3(4,u)$.

(1) Rewriting:



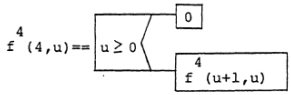
(2) Simplification: According to the definition of the factorial function, factorial(-1) does not terminate. Thus, this partial computation does not terminate.

Example 4:



Example 4: Partially compute $f^4(4,u)$.

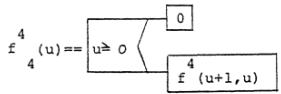
(1) Rewriting:



(2) Simplification:

The same expression as the above.

(3) Tabulation:



Since neither semi-bound calls nor undefined functions are included in the expression, the partial computation terminates.

Termination

- Does not go into the details
- Shows that for “practical” use cases it should terminate
- Cites theoretical works (*e.g.* Ershov)

Theory of Partial Computation

- In the 1930's Turing, Church, Kleene proposed several computational models and clarified the mathematical meanings of mechanical procedure.
- *e.g.* Turing machines, lambda expressions, and partial recursive functions.
- Research on *computability*, *i.e., computational power* of the models, not complexity or efficiency

s_n^m theorem

- Appears in Kleene s_n^m theorem (parameterization theorem, iteration theorem).
- $\varphi_x^{(k)}$ recursive function of k variables with *Gödel number* x;
- then for every $m \geq 1$ and $n \geq 1$ there exists a *primitive recursive function* s such that for all $x, y_1, \ldots, y_m$

$$\lambda z_1, \ldots, z_n \, \varphi_x^{(m+n)}(y_1, \ldots, y_m, z_1, \ldots, z_n) = \varphi_{s(x,y_1,\ldots,y_m)}^{(n)}$$

The third equation of partial computation (α_α) is also used in the proof of Kleene's recursion theorem

Programming Models

- **Turing machines** and **partial recursive functions** were formulated to describe total computation
- **Church's lambda expression** was based upon partial computation

$$f(5, u) \text{ with } u \text{ undefined, yields } f_5(u)$$

Usage in LISPS

"Implementation of a projection machine and its application to real world problems started in the 1960's after the programming language LISP began to be widely used"

GraalVM

Practical Partial Evaluation for High-Performance Dynamic Language Runtimes

Thomas Würthinger[*] Christian Wimmer[*] Christian Humer[*] Andreas Wöß[*]
Lukas Stadler[*] Chris Seaton[*] Gilles Duboscq[*] Doug Simon[*] Matthias Grimmer[†]

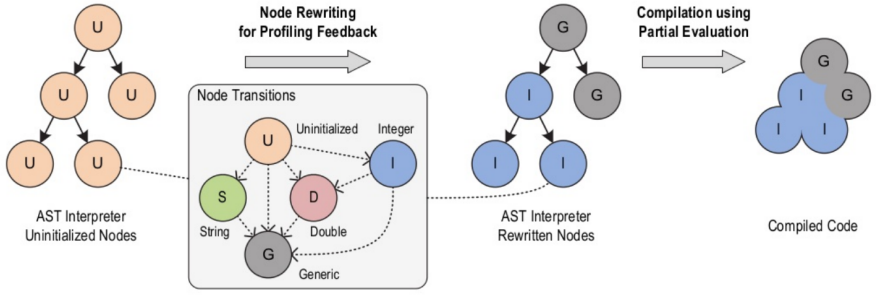
[*]Oracle Labs [†]Institute for System Software, Johannes Kepler University Linz, Austria
{thomas.wuerthinger, christian.wimmer, christian.humer, andreas.woess, lukas.stadler, chris.seaton,
gilles.m.duboscq, doug.simon}@oracle.com matthias.grimmer@jku.at

We implement the language semantics only once in a simple form: as a language interpreter written in a managed high-level host language. Optimized compiled code is derived from the interpreter using partial evaluation. This approach and its obvious benefits were described in 1971 by Y. Futamura, and is known as the *first Futamura projection*. To the best of our knowledge no prior high- performance language implementation used this approach.

We implement the language semantics only once in a simple form: as a language interpreter written in a managed high-level host language. Optimized compiled code is derived from the interpreter using partial evaluation. This approach and its obvious benefits were described in 1971 by Y. Futamura, and is known as the *first Futamura projection*. To the best of our knowledge no prior high- performance language implementation used this approach.

We believe that a simple partial evaluation of a dynamic language interpreter cannot lead to high-performance compiled code: if the complete semantics for a language operation are included during partial evaluation, the size of the compiled code explodes; if language operations are not included during partial evaluation and remain runtime calls, performance is mediocre. To overcome these inherent problems, we write the interpreter in a style that anticipates and embraces partial evaluation. The interpreter specializes the executed instructions, e.g., collects type information and profiling information. The compiler speculates that the interpreter state is stable and creates highly optimized and compact machine code. If a speculation turns out to be wrong, i.e., was too optimistic, execution transfers back to the interpreter. The interpreter updates the information, so that the next partial evaluation is less speculative.

We believe that a simple partial evaluation of a dynamic language interpreter cannot lead to high-performance compiled code: if the complete semantics for a language operation are included during partial evaluation, the size of the compiled code explodes; if language operations are not included during partial evaluation and remain runtime calls, performance is mediocre. To overcome these inherent problems, we write the interpreter in a style that anticipates and embraces partial evaluation. The interpreter specializes the executed instructions, e.g., collects type information and profiling information. The compiler speculates that the interpreter state is stable and creates highly optimized and compact machine code. If a speculation turns out to be wrong, i.e., was too optimistic, execution transfers back to the interpreter. The interpreter updates the information, so that the next partial evaluation is less speculative.


U
U
U
U
U
AST Interpreter
Uninitialized Nodes
Node Rewriting
for Profiling Feedback
Node Transitions
U
Uninitialized
Integer
I
S
String
D
Double
G
Generic
G
I
G
I
I
AST Interpreter
Rewritten Nodes
Compilation using
Partial Evaluation
G
I
G
I
I
Compiled Code


FUTAMURA

References

- Würthinger et al. 2017, Practical Partial Evaluation for High-Performance Dynamic Languages, *PLDI'17*
- Šelajev 2018, GraalVM: Run Programs Faster Anywhere, *GOTO Berlin 2018*
- Bolz et al. 2011, Allocation Removal by Partial Evaluation in a Tracing JIT, *PEPM'11*
- Stuart 2013, Compilers for Free, *RubyConf 2013*
- Cook and Lämmel 2011, Tutorial on Online Partial Evaluation, *EPTCS'11*