Introduction to Partial Evaluation

Compose Project

IRISA / University of Rennes 1- INRIA

March 1998

Goal of Partial Evaluation

- Fact: many programs have layers of interpretation.
- Action: removal of interpretation layers.

Example: Text Formatting

- A procedure that produces formatted text with respect to
  - A control string.
  - A sequence of values.
- It exists in most programming languages
  - `format` in Scheme and Common Lisp.
  - `printf` in C.
- Interpretation layer: the control string

Example: mini_printf

```
void mini_printf(char* fmt, int *value)
{
  int i;

  for (i=0 ; *fmt != '\0' ; fmt++)
    if(*fmt != '%')
      putchar(*fmt);
    else
      switch(*++fmt)
      {
        case 'd': putint(value[i++]); break;
        case '%': putchar('%'); break;
        default: error(); break;
      }
}
```

Example: `mini_printf`

Specialization w.r.t. control string "n = %d"

```
void mini_printf_fmt(int* value)
{
  putchar('n');
  putchar(' ');
  putchar('=');
  putchar(' ');
  putchar(value[0]);
}
```

Assessment

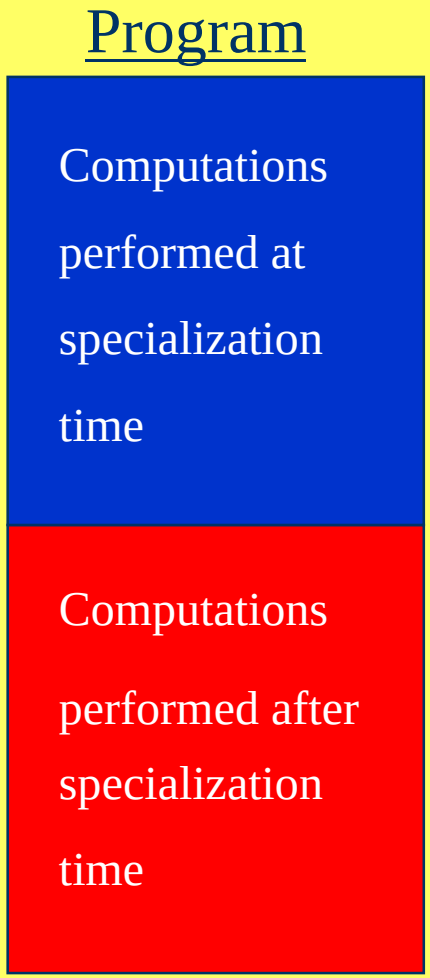
- For any sequence of integer values `values`

```
mini_printf_fmt(values)
=
mini_printf("n = %d", values)
```

- The control string has been interpreted.
- The specialized `mini_printf` is faster than the original procedure.

Partial Evaluation: Summary

- Program specialization.
- Automatic transformation.
- Systematic transformation.
- Interpretation removal.
- Faster specialized programs (usually).
- Smaller specialized programs (sometimes).



Partial Evaluation: Range of Applications

- Operating systems.
- Graphics.
- Scientific computing.
- Software engineering.
- Simulation.
- ...

Outline of the Rest of this Introduction

- The Basics of Partial Evaluation.
- Disclaimers.
- Compile-Time *vs* Run-Time Specialization.
- Program *vs* Data Specialization.
- State of the Art.
- Related Techniques.

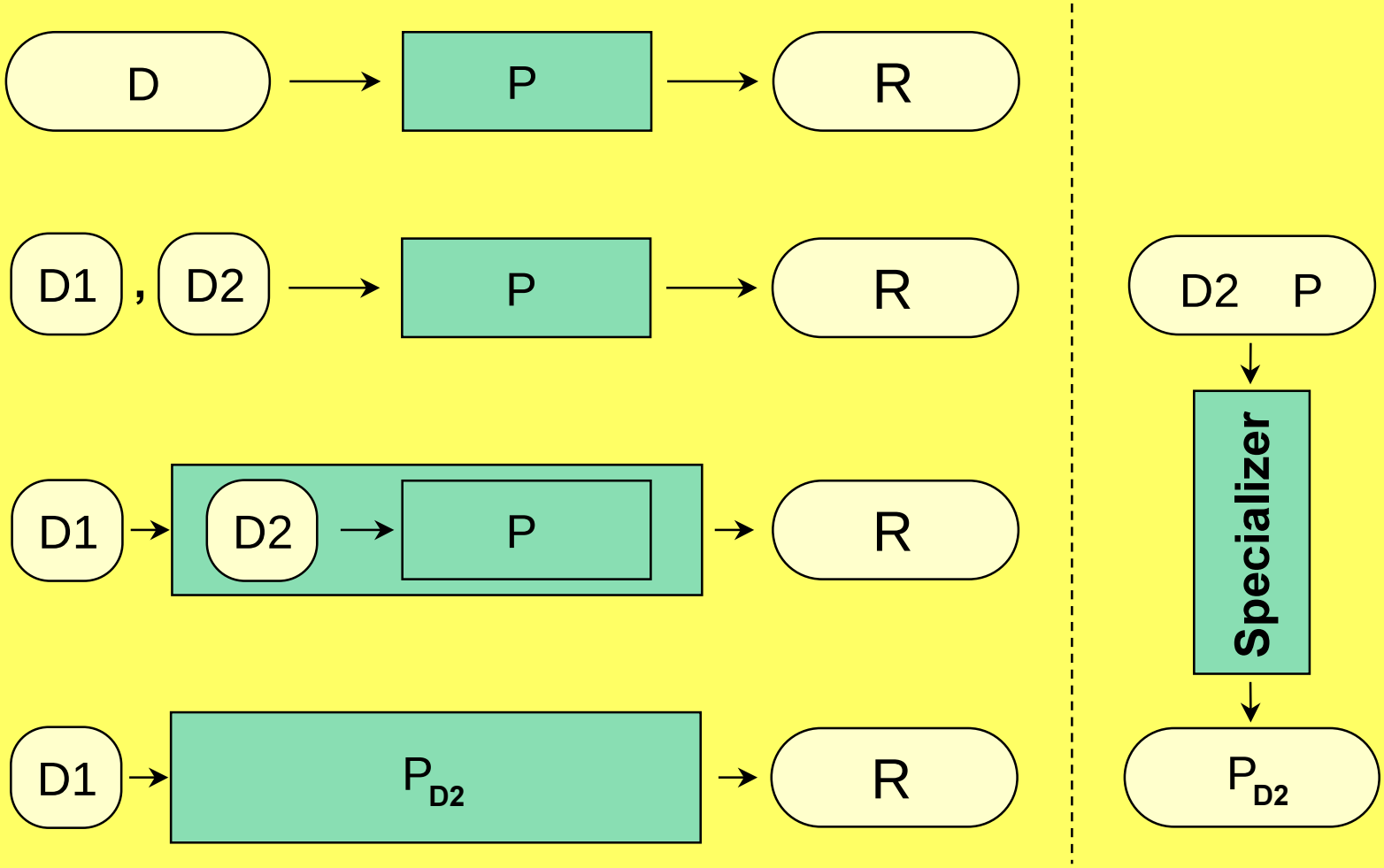
The Basics of Partial Evaluation

- Partial Evaluation: What?
- Partial Evaluation: Why?
- Partial Evaluation: When?
- Partial Evaluation: How?

Partial Evaluation: What?

- A form of program specialization.
- Exploit specialization information.
- Automatic process.

Partial Evaluation



Specialization Information

- Partial input (configuration parameters, usage patterns, ...).
- Properties on inputs (size, shape, ...).
- Abstract data types (sequence as lists, vectors, …).
- Hardware features.
- ...

Partial Evaluation: Why?

Genericity	*vs*	Performance
Parameterized, Modular, Re-usable, Platform-independent		Instantiated, Monolithic, Specific, Machine-dependant optimizations

Examples of generic structures:

software layers, simulators, interpreters, ...

Genericity and Performance?

- Manual optimization.
- Application/code generators.
- Partial evaluation.

Partial Evaluation: When?

Information available at various stages:

- Configuration.
- Compilation.
- Linking.
- Execution (sessions).

Partial Evaluation: How?

- How does partial evaluation work?
- Impact on the degree of specialization?
  - Parallel compilation.
  - Type checking.

How Does Partial Evaluation Work?

Outline:

- Constant propagation.
- Code folding.
- Procedure unfolding (inlining).
- Procedure specialization (cloning/customization).
- Loop unrolling.
- Dead-code elimination.

Constant Propagation

Constants are propagated as much as possible throughout all syntactic constructs.

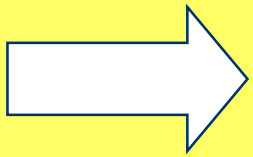
- Intra-procedural propagation.
- Inter-procedural propagation.
- All data types (pointers, arrays, structures, …).

Example: Constant Propagation Throughout Assignments

Values of early bound variables are propagated at specialization time

(assuming no redefinition of `x` before it is used).

```
x = 5;
...
y = x;
```



```
x = 5;
...
y = 5;
```

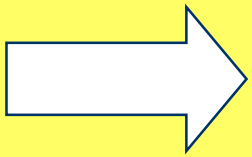
Code Folding

- Syntactic constructs are folded if their evaluation solely depends on available data.
- This operation mimics to the standard evaluation semantics.

Folding of Arithmetic Operations

Early bound computations are performed at specialization time.

```
x = 5;
...
y = x + 10;
```



```
x = 5;
...
y = 15;
```

Procedure Unfolding

- Replacing a procedure call by its body.
- Substituting formals by actuals.

```
f(v, w);
...
void f(int x, int y)
{
  if(y)
    update(x, y);
  else
    update(x,-1);
}
```

```
{
  if(w)
    update(v,w);
  else
    update(v,-1);
}
...
void f(int x, int y)
{...} /* unchanged */
```

Procedure Specialization

- Propagating constant actuals into the called procedure.
- Creating a specialized version of the called procedure.

```
...
f(v, 0);
...
void f(int x, int y)
{
  if(y)
    update(x, y);
  else
    update(x,-1);
}
```

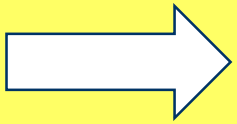
```
...
f_1(v);
...
void f_1(int x)
{
  update(x,-1);
}
```

Procedure Specialization

Specialized versions are given a unique name.

```
f(v, 0);
...
f(v, 0);

void f(int x, int y)
{
  ...
}
```



```
f_1(v);
...
f_1(v);

void f_1(int x)
{
  ...
}
```

Loop Unrolling

Unroll the loop when the test is determined early.

```
for (i=0; *fmt !='\0'; fmt++)
  if(*fmt!='%')
    putchar(*fmt);
  else
   switch(*++fmt)
   {
     case 'd':
       putint(value[i++]);
       break;
     case '%':
       putchar('%'); break;
     default:
       error() ; break;
   }
```

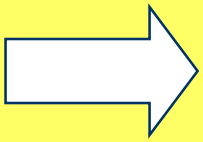
```
/*fmt = "n = %d" */
{
  putchar('n');
  putchar(' ');
  putchar('=');
  putchar(' ');
  putchar(value[0]);
}
```

Dead-Code Elimination

Eliminating code whose computated results or effects are not used.

(assuming `debug` is false).

```
if(debug)
{
 printf("entering ...");
 ++cnt_debug;
}
/* continuing work */
...
```



```
/*continuing work*/
  ...
```

Degree of Specialization

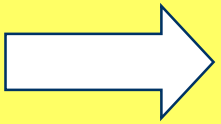
Outline:

- Amount and kind of data available.
- Partial evaluation strategies.
- Structure of the source program.

Amount of Data Available: Complete Data (1/2)

- All the computations can be performed at specialization time.
- Specializing `f` w.r.t. `1` and `2`
- Result: a constant function.

```
int f(int i,int j)
{
   return i+j;
}
```

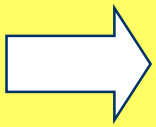


```
int f_1()
{
   return 3;
}
```

Amount of Data Available: Complete Data (2/2)

- Some computations cannot be performed at specialization time.
- Assuming `printf` cannot be evaluated at specialization time.
- The residual program contains a call to `printf`.

```
int f(int i, int j)
{
  printf("i=%d / j=%d",
         i , j);
  return i+j;
}
```

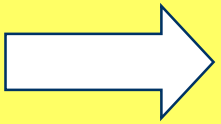


```
int f_1()
{
  printf("i=%d / j=%d",
         1 , 2);
  return 3;
}
```

Amount of Data Available: No Data (1/2)

When no data are available, the program is reconstructed.

```
int f(int i,int j)
{
   return i+j;
}
```



```
int f(int i,int j)
{
   return i+j;
}
```

Amount of Data Available: No Data (2/2)

Yet, some input-independent optimizations can be performed.

- Procedure unfolding.
- Use of program constants.

Amount of Data Available: Partial Data

- When some (but not all) of the data are available, the program is
  - Partly evaluated.
  - Partly reconstructed.
- Mixed computations.

Approaches To Partial Evaluation

Outline:

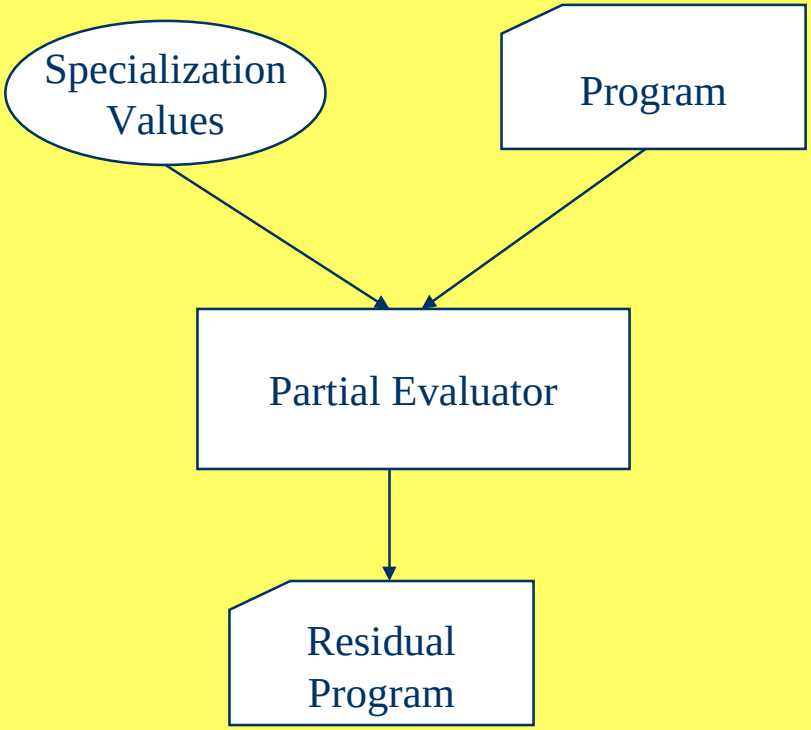
– Online partial evaluation.

– Offline partial evaluation.

Online Partial Evaluation: Principles

- All decisions made on the fly.
- Based on specialization values.
- Interpretation of specialization.

Online Partial Evaluation



Online Partial Evaluation: Example

Assuming `test` may be known or unknown at specialization time:

3 possible cases.

```
if (test)
  x=3;
else
  x=unknown_value;
do_this(x+10);
/* continue work */
```

1.

```
/*if test known and true*/
do_this(13);
/* continue work */
```

2.

```
/*if test known and false*/
do_this(unknown_value+10);
/* continue work */
```

3.

```
/* if test unknown no change*/
...
/* continue work */
```

Online Partial Evaluation: Assessment

- Specialization is precise.
- The degree of specialization is difficult to predict.
- On-the-fly decision causes overhead for repeatedly processed code when the specialization context is unchanged.
- How to perform specialization efficiently at run time?

Terminology

In the program transformation context,

– *known* data refers to available data,

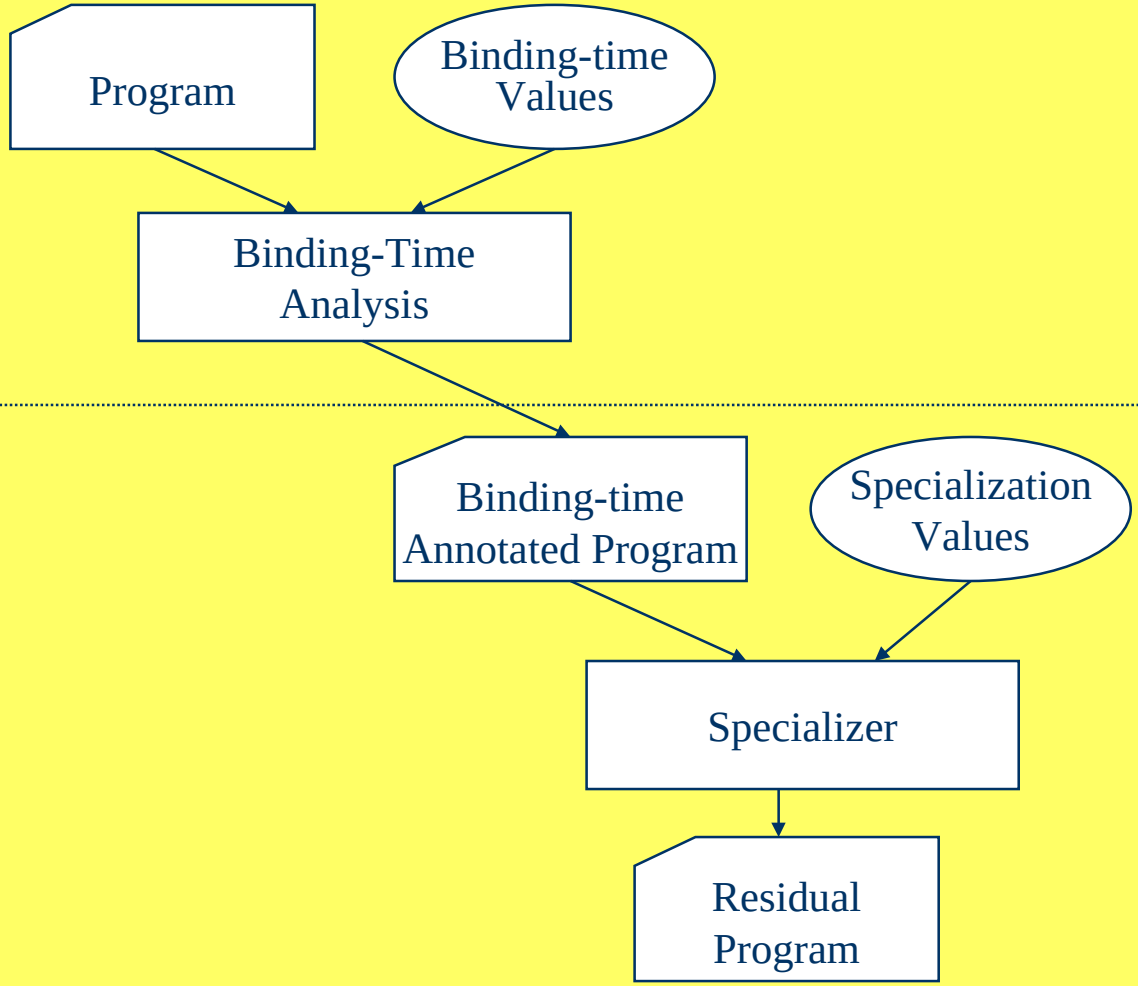
– *unknown* data refers to unavailable data.

Offline Partial Evaluation: Principles

- Two phases:
  - Binding-time analysis (BTA): determining known computations.
  - Specialization: performing transformations based on known (actual) data and binding-times.
- The specialization is “compiled”.

Offline Partial Evaluation

- Preprocessing: once for a binding-time description.
- Specialization: for all specialization values.



Binding-Time Analysis

- Given
  - a program,
  - a binding-time description of its input.

  BTA determines the computations that can be performed at specialization time.
- The binding-time properties produced by the BTA are valid relative to the binding-times of the input program.

Binding-Time Analysis Example: `mini-printf`

```
void mini_printf(char* fmt, int *value)
{
   int i;

   for (i=0 ; *fmt != '\0' ; fmt++)
   if(*fmt != '%')
     putchar(*fmt);
   else
     switch(*++fmt)
     {
       case 'd': putint(value[i++]); break;
       case '%': putchar('%'); break;
       default: error() ; break;
     }
}
```

Specialization

- It exploits binding-time information.
- It determines and performs the program transformation, given specialization values.

Offline Partial Evaluation: Assessment

- The specialization process is compiled: Faster than online partial evaluation.
- Easier to perform specialization both at compile time and run time.
- The specialization is approximative: Driven by binding-time properties, not by specialization values.

Offline Partial Evaluation: Example of Approximation

Assuming `test` is known at specialization time.

```
if (test)
  x=3;
else
  x=unknown_value;
do_this(x+10);
/* continue work */
```

Terminology

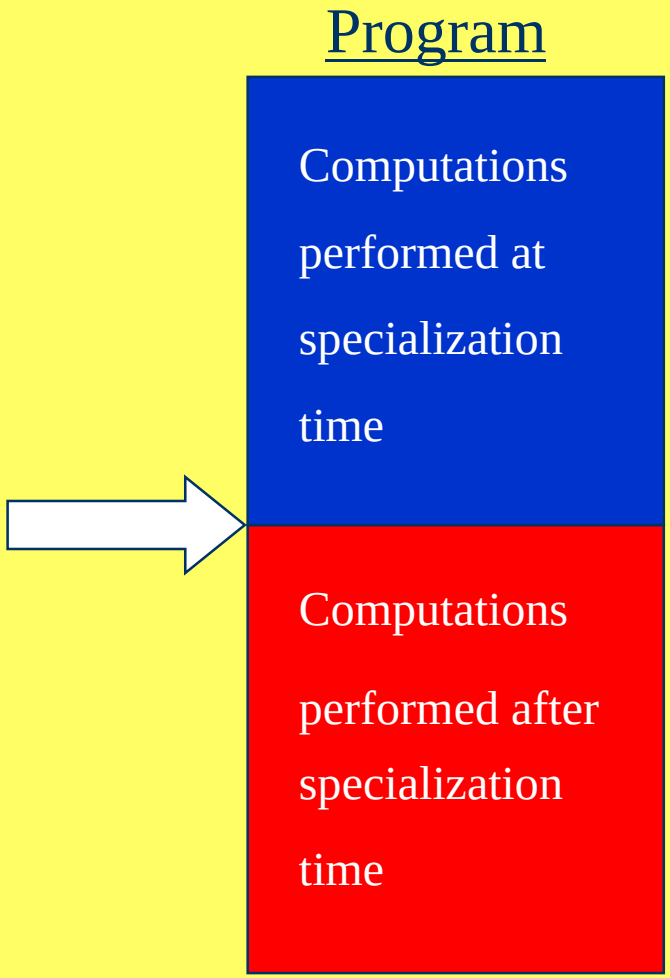
In the binding-time analysis context,

– *static* refers to available data,

– *dynamic* refers to unavailable data.

Online *vs* Offline Partial Evaluation

- Online partial evaluation:
  - Partitioning between known and unknown computations is determined on the fly.
  - Partitioning depends on known/unknown program inputs and their specific values.
  - Online partial evaluation is more precise.
- Offline partial evaluation:
  - Partitioning between static and dynamic computations is fixed.
  - Partitioning only depends on the static/dynamic program inputs.
  - Offline partial evaluation is more predictable.



Disclaimers

- Partial evaluation is simple and automatic but it is not magic.
- Potential weaknesses.

Outline:

- Limited algebraic transformations.
- Code explosion.
- Unsafe static computations.
- Non-termination.

Limited Algebraic Transformations

- Few, if any, algebraic transformations are offered.
- This strategy is aimed at making the partial evaluation process:
  - Simple.
  - Automatic.
  - Predictable.
  - Efficient.

Limited Algebraic Transformations: Example

Assuming `x=unknown, y=4, z=5`

```
... (x + y) + z ...
```

```
... (x + 4) + 5 ...
```

Unless the partial evaluator is instructed that addition is associative.

```
... x + (y + z) ...
```

```
... x + 9 ...
```

Partial evaluators usually do not include algebraic transformations.

Space Explosion

It can happen with

- Procedure unfolding.
- Procedure specialization.
- Loop unrolling.

Unsafe Static Computations

For a conditional statement

```
if (unknown_test)
  then_stmt;
else
  else_stmt;
```

Typically a partial evaluator processes both branches speculatively.

Unsafe Static Computations

Speculative processing of conditionals may be

- Unnecessary:
  - » The specialized branch may never be executed.
- Unsafe:
  - » A branch may loop.
  - » A branch may perform a side-effect (e.g., an error).

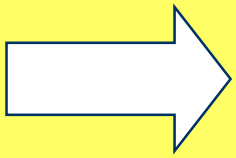
Problems of Non-Termination

Outline

- Infinite unfolding.
- Infinite specialization.
- Infinite unrolling.

Infinite Unfolding

```
f(i, j);
...
void f(int x, int y)
{
  something;
  if(test)
    f(v, w);
  else
    return;
}
```



```
{
  something;
  if(test)
    {
      something;
      if(test)
      {
        something;
        if(test)
        {
          ...
```

Infinite Specialization

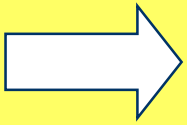
```
f(i, j);
...
void f(int x, int y)
{
  something;
  if(test)
    f(x, ++y);
  else
    return;
}
```

```
f_1(i);
...
void f_1(int x)
{
  something;
  if(test)
    f_2(x);
  else
    return;
}
void f_2(int x)
{
  something;
  if(test)
    f_3(x);
  else
    return;
}
...
```

Infinite Unrolling (1)

Dynamic test of a loop being unrolled.

```
...
while(test)
  something;
```

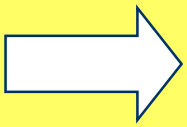


```
...
something;
something;
something;
...
```

Infinite Unrolling (2)

Static test of a forever loop with dynamic test to exit it (`break`).

```
...
while(true)
  something_else;
```



```
...
something_else;
something_else;
something_else;
...
```

When To Specialize?

- Compile-time specialization.
- Run-time specialization.

Compile-Time Specialization

- When the specialization values:
  - are known prior to run time,
  - do not change at run time,
  - change at run time but can only have few variations which are known at compile time.
- Two binding times:
  - static = compile time,
  - dynamic = run time.

Compile-Time Specialization

- Partial evaluation can either be online or offline.
- It essentially consists of the transformations described earlier.
- It is done prior to running the residual program: no cost.

Run-Time Specialization

- Specialization values are available at run time.
- They may not be valid throughout the execution.
- Two binding times within the run time:
  - static = early,
  - dynamic = late.
- Specialized code is built at run time.

Run-Time Specialization: Cost/Benefits

- Run-time specialization process:
  - Good quality code / slow.
  - Poor quality code / fast.
- For a given run-time specialized code:
  - Amortizing the cost of specialization.
    - » Performance improvement.
    - » Number of uses.

Another Form of Specialization: Data Specialization

A form of specialization complementary to program specialization.

Limitations of Program Specialization

- Code explosion (loop).
- Inappropriate for some programs (large data sets).
- Complex program transformation process.

Principles of Data Specialization (1/2)

- Expensive computations which are known are stored in
  - *program* = program specialization,
  - *data* = data specialization.
- Optimization of the data flow.
- No optimization of the control flow.

Principles of Data Specialization (2/2)

Basic components.

- Cache:
  - » Data structure where known values are stored.
- Loader -- known computations:
  - » Performs the known computations.
  - » Stores the resulting values in the cache.
- Reader -- unknown computations:
  - » Uses the values contained in the cache.
  - » Performs the unknown computations.

Example (1/3)

- Known variables: `i` and `max_i`.
- Unknown variable: `j`.
- Data specialization: caching the expensive computations on `i` and `max_i`.

```
...
for(i=0 ; i < max_i ; i++)
  for(j=0 ; j < max_j ; j++)
    f(i, max_i, j);
...
```

Example (2/3)

```
int f(int x, int y, int z)
{
  return (x*y + z) / z;
}
```

Expensive known computations

```
void f_load(int x, int y,
            int *cache)
{
  *cache = x * y;
}
```

```
int f_read(int z, int cache)
{
  return(cache + z)/z;
}
```

Example (3/3)

```
...
{
  int cache[MAX];

  for(i=0 ; i<max_i ; i++)
  {
    for(j=0 ; j<max_j ; j++)
      f_load(i, max_i, cache+j);

    for(j=0 ; j<max_j ; j++)
      f_read(j, cache[j]);
  }
...
```

Specializing Shaders

Geometric renderer

- Scene:
  - » Objects, orientation, coordinates.
  - » Shades, textures, position of lights, colors.
- Using a pre-computed geometric image.
  - » Varying parameters: shading, reflectance.
  - » Non-varying parameters: eye point, orientation, object coordinates.
- Data specialization.
  - » Intermediate results pre-computed, cached, and re-used.
  - » Speedup factor: 60

State of the Art

Outline:

- Languages.
- Partial evaluators.
- Applications.

State of the Art: Languages

- Applicative languages (Lisp, Scheme, ML).
- Logic languages (Prolog, CLP).
- Imperative languages (Pascal, Fortran, C).
- Object-oriented languages (simple OO dialect).
- Concurrent languages (formally).

State of the Art: Partial Evaluators

- Applicative languages:
  - Fuse: pure Scheme, online, [Weise *et al* 1990].
  - Schism: pure Scheme, offline, [Consel 1990].
- Logic languages:
  - Promix: pure Prolog, online [Lakhotia *et al* 1991]
- Imperative languages:
  - FSpec: Fortran subset, offline, [Glueck *et al* 1995].
  - C-Mix: C, offline, [Andersen 1994].
  - Tempo: C, offline, [Consel *et al* 1996].

State of the Art: Applications (1/3)

- Interpreters (compiler generation).
- String matching.
- Pattern matching.

State of the Art: Applications (2/3)

- Numerical programs:
  - Fast Fourier Transform.
  - Convolution.
- Graphics:
  - Ray tracer.
  - Dithering.
- Operating systems:
  - Sun Remote Procedure Call.
  - Chorus Inter-Process Communication.
  - Berkeley Packet Filter.

State of the Art: Applications (3/3)

- Domain-specific languages:
  - Video device drivers.
  - Active networks.
- Software architectures:
  - Selective broadcast.
  - Software layers (RPC).
  - Interpreters (DSL).

Related Techniques

Outline:

- Optimizing compilation.
- Manual Code Generation.
- Dynamic Compilation.

Related Techniques: Optimizing Compilation (1/2)

- Different target programs:
  - Optimizing compilers need to treat a wide variety of program patterns.
  - Partial evaluators focus on generic programs.
- Different constraints:
  - Optimizing compilers offer accurate analyses only if they pay off on common cases.
  - Partial evaluators have a much more narrow spectrum: they offer accurate analyses to enable high-degree of specialization.

Related Techniques: Optimizing Compilation (2/2)

Partial evaluation features, not present in optimizing compilers:

- Intra/inter-procedural propagation of non-scalar constants.
- Partially known data.
- Run-time specialization.
- Providing specialization information.

Related Techniques: Manual Code Generation (1/2)

- Language extensions to allow a user to compute programs at run time.
- Low-level program representation:
  - It enables low-level code optimizations (instruction scheduling, register allocation, ...)
  - Example: DCG [Engler & Proebsting]
- Source-level program representation:
  - It enables source-level optimizations.
  - Example: Tick C [Engler *et al*]: similar to backquote mechanism in Lisp.

Related Techniques: Manual Code Generation (2/2)

- Advantages:
  - Very precise.
  - Very flexible.
- Drawbacks:
  - Error prone.
  - Laborious.

Related Techniques: Dynamic Compilation (1/4)

Postponing part of the compilation process until run time:

- To exploit run-time values:
  - » Algebraic simplifications.
  - » Program specialization.
- To exploit run-time control patterns:
  - » Loop optimization.
  - » Register allocation.
  - » Instruction selection/scheduling.

Related Techniques: Dynamic Compilation (2/4)

Fabius [Leone & Lee]:

- First-order, pure functional language.
- Special-purpose compiler.
- Preprocessing phase includes a form of binding-time analysis.
- Compiler generates:
  - » Instructions to perform static computations.
  - » Instructions to generate the code at run time for the dynamic computations.
  - » Instructions to perform optimizations.

Related Techniques: Dynamic Compilation (3/4)

DyC [Auslander *et al*]

- C language and declarations (regions and transformations).
- Modified version of the Multiflow compiler.
- Preprocessing phase includes a binding-time analysis.
- Compiler generates:
  - » Instructions to perform static computations.
  - » Binary templates for dynamic computations.
  - » Instructions to fill and assemble templates.

Related Techniques: Dynamic compilation (4/4)

- Cost/Benefit:

  Like run-time specialization.
  - Dynamic compilation:
    - » Highly-optimized code / slow.
    - » Poor quality code / fast.
    - » Architecture dependent.
  - For a given dynamically compiled code.
    - » Amortizing the cost of compilation.
      - Performance improvement.
      - Number of uses.

Summary of the Introduction to Partial Evaluation

- The Basics of Partial Evaluation.
- Disclaimers.
- Compile-Time *vs* Run-Time Specialization.
- Program *vs* Data Specialization.
- State of the Art.
- Related Techniques.