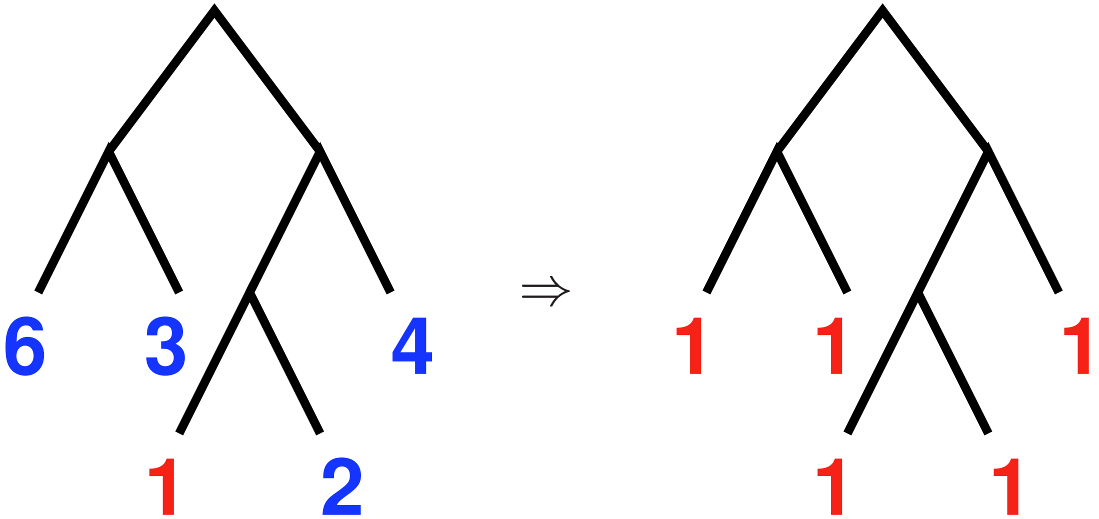
Implementing circularity using partial evaluation

Julia Lawall
DIKU, University of Copenhagen
May 22, 2001

Circularity

Classical example: `repmin` [Bird]

Replace each leaf by the minimum leaf.



Intermediate computation (rebuilding each leaf) depends on part of the final result (a new tree and the minimum leaf value).

Other examples

Compilation: stack *vs.* register allocation

- `{int x = 3; f(&x)}`: `x` must be stack-allocated
- `{int x = 3; g(x)}`: `x` can be register-allocated

Partial evaluation: `Let`-insertion, used to name multiply referenced dynamic expressions

Run-time code generation:
Local optimizations based on global properties (code size, etc).

Lazy implementation [Bird]

Tree traversal (`rm`):

- Inputs: Tree and new leaf value
- Outputs: New tree and computed minimum leaf value

```
rm (Tip n) m = (Tip m, n)
rm (Leaf t1 t2) m = ...
```

Main function (`repmin`):

```
repmin t = fst p
  where p = rm t (snd p)
```

Attribute-grammar approach

Attribute grammar:

- A declarative description of a data structure and its properties
- Properties can depend on those of substructures and enclosing structures

Computation of properties implemented by:

- A static scheduler, or
- Translation into a lazy language

Implementation [Kuiper, Swierstra]

$$\begin{aligned} \mathit{Root} &: \mathit{Start} \to \mathit{Tree} \\ \mathit{Start.res} &= \mathit{Tree.res} \\ \mathit{Tree.glob} &= \mathit{Tree.loc} \end{aligned}$$

$$\begin{aligned} \mathit{Tip} &: \mathit{Tree} \to \mathit{Val} \\ \mathit{Tree.res} &= \mathit{Tip}(\mathit{Tree.glob}) \\ \mathit{Tree.loc} &= \mathit{Val.val} \end{aligned}$$

$$\begin{aligned} \mathit{Node} &: \mathit{Tree}_0 \to \mathit{Tree}_1\,\mathit{Tree}_2 \\ \mathit{Tree}_0.\mathit{res} &= \mathit{Node}(\mathit{Tree}_1.\mathit{res}, \mathit{Tree}_2.\mathit{res}) \\ \mathit{Tree}_0.\mathit{loc} &= \mathit{min}(\mathit{Tree}_1.\mathit{loc}, \mathit{Tree}_2.\mathit{loc}) \\ \mathit{Tree}_1.\mathit{glob} &= \mathit{Tree}_0.\mathit{glob} \\ \mathit{Tree}_2.\mathit{glob} &= \mathit{Tree}_0.\mathit{glob} \end{aligned}$$

Imperative implementation

?

Our solution

Language construct:

Persistent variables

Implementation strategy:

Partial evaluation (data specialization),
building on Tempo [Consel et al.]

Overview

1. Persistent variables
2. Partial evaluation (data specialization)
3. Data specialization with persistent variables
4. Comparison with attribute grammars
5. Conclusion

Source language

C extended with:

- `persistent` t `x`:
  Declare a persistent variable `x` of type t.
- `pread(x)`: "Persistent read"
  Access the value of persistent variable `x`.
- `pwrite(x,`e`)`: "Persistent write"
  Set persistent variable `x` to the value of e.

`pread(x)` gives the final value `x` is assigned by any `pwrite(x,`e`)`.

(Semantics in the paper)

repmin with persistent variables

Return multiple values:

```
typedef struct ans { int mn; Tree *tree; } Ans;
```

Traverse tree:

```
void rm(Tree *t, int m, Ans *a) { ... }
```

Main function:

```
Tree *repmin(Tree *t) {
  persistent int minval;
  Ans a;
  rm(t, pread(minval), &a);
  pwrite(minval,a.mn);
  return a.tree;
}
```

Implementation

Stage the program such that:

- All `pwrite`'s are in the first stage.
- All `pread`'s are in the second stage.

Generalizes to multiple stages.

Partial evaluation

A program-staging technology:

- Inputs: Program and description of known values.
- (Binding-time analysis: What is known?)
- First stage: Evaluate known (static) terms.
- Second stage: Evaluate unknown (dynamic) terms.

Variants of partial evaluation

Program specialization:

Specializer

- Evaluates static terms
- Constructs code to evaluate dynamic terms

Data specialization:

Specializer

- Constructs code (loader) to evaluate static terms
- Constructs code (reader) to evaluate dynamic terms
- Loader and reader communicate via a Cache

Data specialization example

Binding-time analysis:

blue = static, red = dynamic

```
int f(int s, int d) {
  s = s + s;
  d = d + s;
  s = s + 1;
  d = d - s;
}
```

Data specialization example, contd.

Loader:

```
int f_ldr(int s, int *cache) {
  s = s + s;
  cache[0] = s;
  s = s + 1;
  cache[1] = s;
}
```

Reader:

```
int f_rdr(int d, int *cache) {
  d = d + cache[0];
  d = d - cache[1];
}
```

Data specialization and persistent variables

Goals:

- `pwrite(x,`e`)` in the first stage.
  Thus, `pwrite(x,`e`)` should be static.
- `pread(x)` in the second stage.
  Thus, `pread(x)` should be dynamic.
- Every `pread(x)` sees the effect of the final `pwrite(x,`e`)`.
  How should this be implemented?

Partial evaluation preserves dataflow

Source-program evaluation (initially $\texttt{s} = 2, \texttt{d} = d$):

`s = s + s;`	$\texttt{s} = 4, \texttt{d} = d$
`d = d + s;`	$\texttt{s} = 4, \texttt{d} = d + 4$
`s = s + 1;`	$\texttt{s} = 5, \texttt{d} = d + 4$
`d = d - s;`	$\texttt{s} = 5, \texttt{d} = d - 1$

Specialized-program evaluation:

`s = s + s;`	$\texttt{s} = 4$		$\texttt{d} = d$
`cache[0] = s;`	$\texttt{s} = 4$	`d = d + cache[0];`	$\texttt{d} = d + 4$
`s = s + 1;`	$\texttt{s} = 5$		$\texttt{d} = d + 4$
`cache[1] = s;`	$\texttt{s} = 5$	`d = d - cache[1];`	$\texttt{d} = d - 1$

Dataflow of persistent variables

Values flow:

- From the end of the loader
- To the beginning of the reader

Solution: transmit values using the data-specialization cache.

Implementation strategy

- `persistent` t `x`: Allocate a location for `x` in the cache.
- `pread(x)`: Access the chosen cache location.
- `pwrite(x,`e`)`: Update the chosen cache location.

(Formalized in the paper)

repmin binding-time analysis

```
Tree *repmin(Tree *t) {
  persistent int minval;
  Ans a;
  rm(t, pread(minval), &a);
  pwrite(minval,a.mn);
  return a.tree;
}
void rm(Tree *t, int m, Ans *a) {
  if (t->type == Tip) {
    a->mn   = t->tipval;
    a->tree = mkTip(m);
  } else ...
}
```

blue = static

red = dynamic

pink = static and dynamic

Data specialization of `repmin`

Loader:

```
void repmin_ldr(Tree *t, int *cache) {
 Ans a;
 int *minval_ptr = cache++;
 cache = rm_ldr(t, &a, cache);
 *minval_ptr = a.mn;
}
```

Reader:

```
Tree *repmin_rdr(int *cache) {
 Ans a;
 int minval = *cache++;
 cache = rm_rdr(minval, &a, cache);
 return a.tree;
}
```

Comparison with attribute grammars

- Both approaches use analyses to schedule computations. Attribute-grammar (AG) analyses more powerful w.r.t. circularity.
- AG focuses on a description of the input structure. Partial evaluation focuses on a program.
- AG uses a declarative notation. Our approach treats an imperative language.
- AG scheduler detects circularity automatically. In our approach, circular references must be annotated to avoid C's flow-sensitive semantics.

Conclusion

- A new approach to implementing circularity:
  Persistent variables combined with data specialization
  - Allows circularity in imperative programs
  - Implemented in Tempo
  - Used in ongoing work on run-time specialization
- A new perspective on the relationship between partial evaluation and attribute grammars