Data Dependences and Advanced Induction Variables Detection

Sebastian Pop

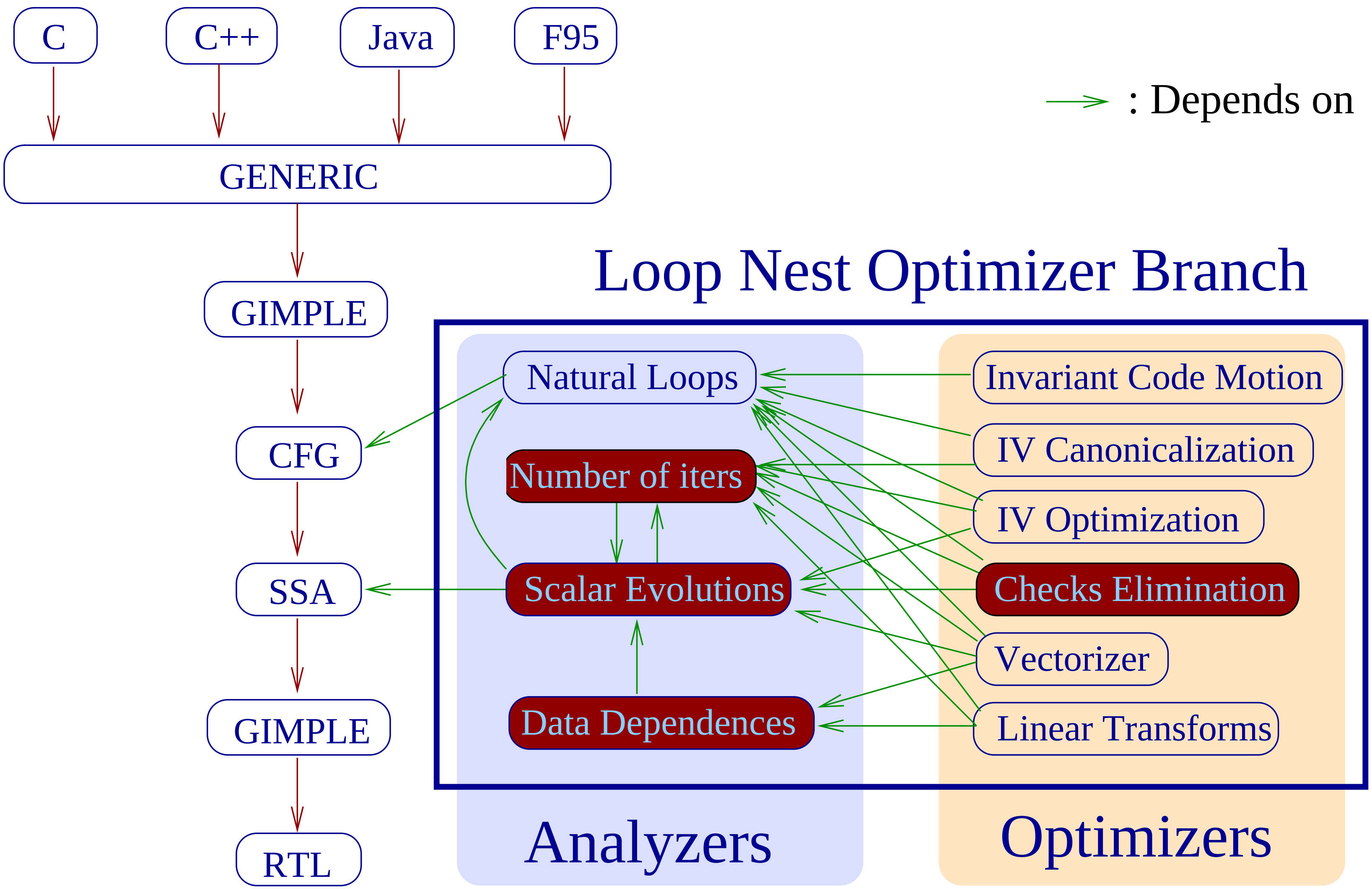
pop@cri.ensmp.fr

Centre de Recherche en Informatique (CRI)

École des mines de Paris

France

Overview of LNO



Data Dependence?

```
DO I=4,8
 DO J=3,8
  A(I, J) = A(I-3, J-2) + 1
 END DO
END DO
```

At iteration I = 7, J = 4,
A(7,4) = A(7-3,4-2)+1
so,
A(4,2) **must** be computed **before** A(7,4).

This **data dependence** can be summarized by a mathematical abstraction, like the **distance** vector:

$$Dist = \begin{pmatrix} 3 \\ 2 \end{pmatrix}$$

Computing Data Dependences

```
DO I = 0, N
  T[f(I)] = ...
  ... = T[g(I)]
END DO
```

Are the elements of `T` accessed several times?
i.e. are there some values $x, y \in [0, N]$ such that:

$$f(x) = g(y),\ f(x) = f(y) \text{ or } f(x) = g(y)$$

$\longrightarrow$ Need a description of the values of f and g.

Induction Variables (IV)

```
DO I = 0, N
  T[a] = ...
  ... = T[b]
END DO
```

- Variables a and b are **induction variables**: their values may change with successive `I` values.
- Goal: describe scalar variables in loops
  - give the successive values (when possible),
  - give a range or an envelope of values.

Chains of Recurrences

- Representation of successive values in loops using a form called **chains of recurrences**.
- For instance, the chain of recurrence

$$\{1, +, 3\}$$

represents the values of a in the program:

```
a = 1
DOFOREVER
  a = a + 3
END DO
```

Analyzing SSA Programs

```
a = 1
DOFOREVER
   a = a + 3
END DO
```

$\xrightarrow{SSA\ representation}$

```
a = 1
loop
   b = phi (a, c)
   c = b + 3
endloop
```

- Use-def links,
- Phi nodes at control flow junctions.

Induction Variable Analysis

- Value of a variable at each iteration of a loop,
- Analyze on demand,
- Store intermediate results,
- Algorithm:
  1. Walk the use-def edges, find a SCC,
  2. Reconstruct the update expression,
  3. Translate to a chain of recurrence,
  4. (optional) Instantiate parameters.

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

The initial condition is a definition outside the loop.

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

Depth-first walk the use-defs to a loop-phi node:

$$c \longrightarrow f \longrightarrow e \longrightarrow d$$

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$d \neq c$, walk back, search for another loop-phi:

$$d \to e \to f$$

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

Found the starting loop-phi. The SCC is:

$$c \longrightarrow f \longrightarrow c$$

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

Reconstruct the update expression:

$$c + e$$

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$$c = phi\ (a,\ c + e)$$

$$c \rightarrow \{a, +, e\}$$

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$$c \rightarrow \{a, +, e\} \xrightarrow{Instantiate} Optional \cdots$$

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$$c \rightarrow \{a, +, e\} \xrightarrow{Instantiate} \{3, +, e\}$$

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$$c \rightarrow \{a, +, e\} \xrightarrow{Instantiate} \{3, +, e\}$$

$$e \rightarrow d + 7$$

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$$c \rightarrow \{a, +, e\} \xrightarrow{Instantiate} \{3, +, e\}$$

$$e \rightarrow d + 7$$

$$d \rightarrow \{1, +, 5\}$$

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$$c \rightarrow \{a, +, e\} \xrightarrow{Instantiate} \{3, +, e\}$$

$$e \rightarrow \{8, +, 5\}$$

$$d \rightarrow \{1, +, 5\}$$

Example

```
a = 3
b = 1
loop
  c = phi (a, f)
  d = phi (b, g)
  if (d > 123) goto end
  e = d + 7
  f = e + c
  g = d + 5
endloop
end:
```

$$c \rightarrow \{a, +, e\} \xrightarrow{Instantiate} \{3, +, 8, +, 5\}$$

$$e \rightarrow \{8, +, 5\}$$

$$d \rightarrow \{1, +, 5\}$$

Summary

From the SSA program:

```
loop_1
  f = phi (init, f + step)
endloop
```

Extract the symbolic evolution:

$$\begin{aligned} f(x) &\rightarrow \{init, +, step\}_1(x) \\ x &= 0, 1, \ldots, N \end{aligned}$$

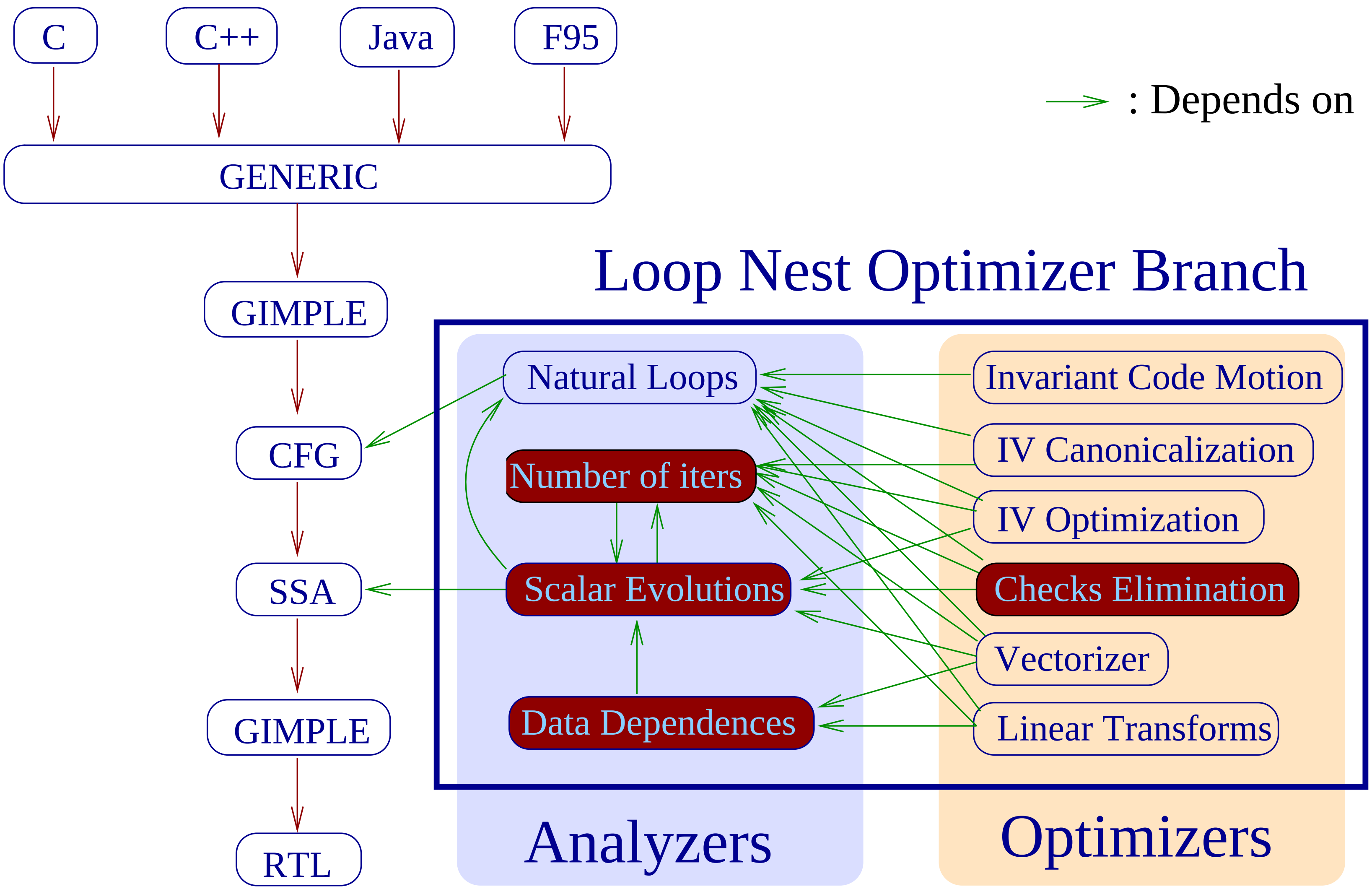
Optionally, instantiate the parameters $init$ and $step$.

Applications

Why another IR?

- information about the evolution of a scalar variable in a loop cannot be represented in SSA
- other analyzers need this information:
  - data dependence testers,
  - number of iterations.

Applications



Number of iterations

```
loop
  ...
  if (a > b) goto end
  ...
endloop
end:
```

1. Find the evolution of a and b,
2. Call the niter solver. The result is:
   - an integer constant,
   - a symbolic expression.

Condition Elimination

Algorithm:

1. compute the number of iterations
   - in the loop,
   - in the then clause,
   - in the else clause.
2. when all the iterations fall in one of the branches, eliminate the unused branch.

CCP after Loops

```
loop
  ...
  a = ...
  ...
endloop
 a = value after crossing the loop
```

Algorithm:

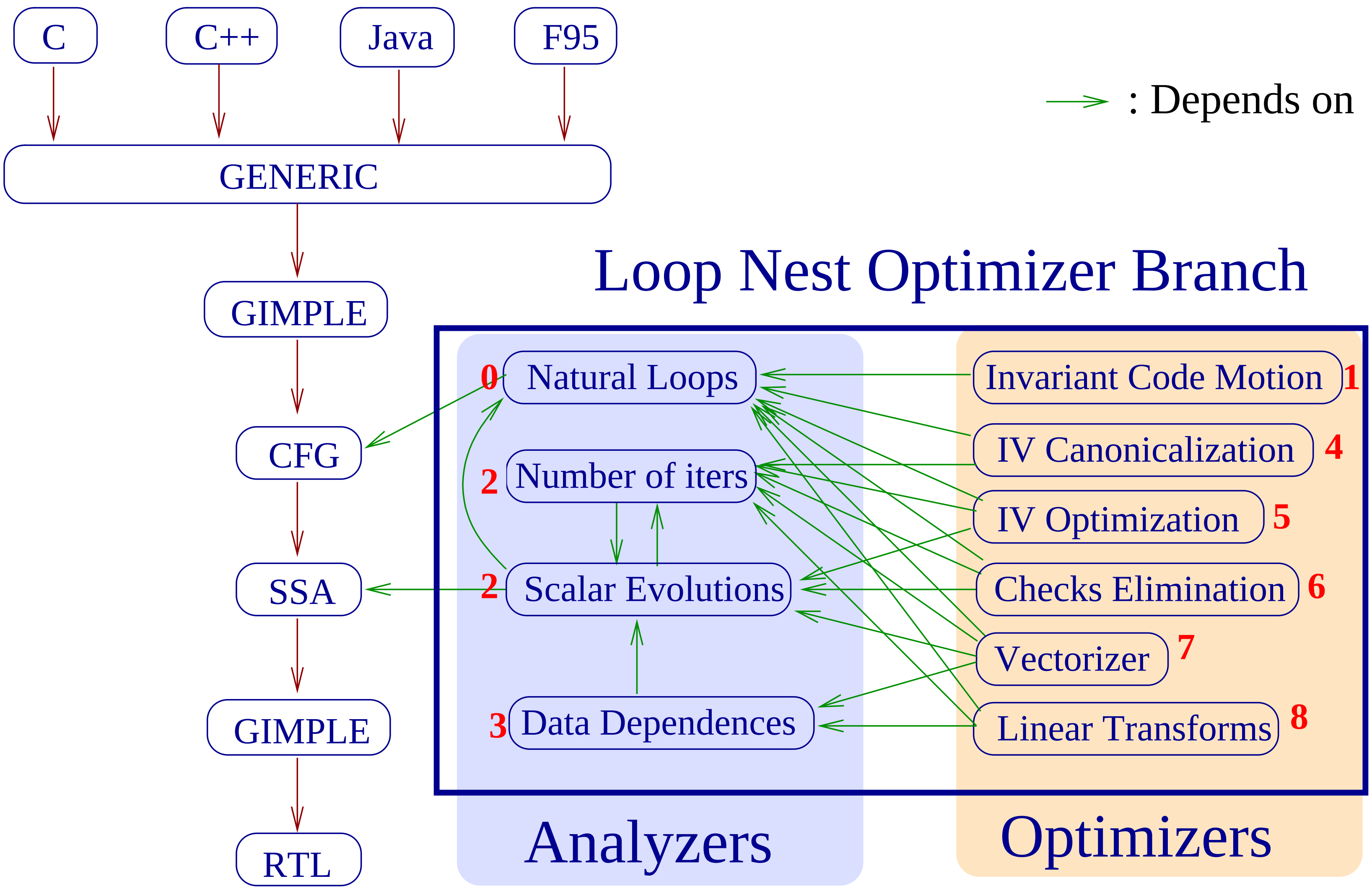
1. compute the value of a scalar variable after crossing a loop,
2. assign this value to the variable after the loop,
3. call the constant propagation optimizer.

Conclusion

The LNO branch contains:

- a fast algorithm for analyzing variables in loops,
- the classic Banerjee data dependence testers,
- induction variable optimizations,
- the linear loop transformations,
- the vectorizer.

Merge plan


C
C++
Java
F95
GENERIC
GIMPLE
CFG
SSA
GIMPLE
RTL
: Depends on
Loop Nest Optimizer Branch
0
Natural Loops
2
Number of iters
2
Scalar Evolutions
3
Data Dependences
Invariant Code Motion
1
IV Canonicalization
4
IV Optimization
5
Checks Elimination
6
Vectorizer
7
Linear Transforms
8
Analyzers
Optimizers