Towards application of supercompilation and metacomputation to high performance computing

Supercompilation 40 years later

Andrei V. Klimov

Head of Program Analysis and Transformation Sector
Keldysh Institute of Applied Mathematics
Russian Academy of Sciences
Moscow, Russia

NII Shonan Meeting, 21 May 2012, Japan

Outline

- On terminology
  - Do I understand term “staging” correctly?
  - Term “metacomputation”
- History of supercompilation
- Java Supercompiler JScp
- Conclusions

Do I understand term “staging” correctly?

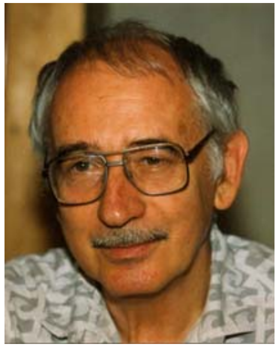
- Program specialization ≠ Staging
  - partial evaluation, supercompilation...
  - equivalence transformation
  - no syntax for specialization time computations
  - user can choose variants of specialization
  - user cannot violate equivalence
- Staging
  - syntax for computation at different stages
  - some means for user-defined transformations
  - the user can violate equivalence
- Macro generation ≠ Staging
  - special syntax for macro definitions = text processing
  - the user can do anything

Term “metacomputation”

- Metacomputation
  - umbrella term for non-trivial program manipulation
    - “semantic-based program manipulation”
  - program specialization
    - offline/online partial evaluation
    - supercompilation
      - close to online partial evaluation
    - partial deduction (for logic programming languages)
      - close to supercompilation
    - ...
  - staging
  - program inversion
  - ...

Metacomputation Workshops in Pereslavl-Zalessky, Russia

organized by Program Systems Institute, RAS



Valentin Turchin
(1937-2010)

- 2008 July 2-5
  - First International Workshop on Metacomputation in Russia
    http://meta2008.pereslavl.ru/
- 2010 July 1-5
  - Second International Workshop on Metacomputation in Russia
    http://meta2010.pereslavl.ru/

  invited speakers
  - Neil Jones
  - Simon Peyton Jones
- **2012 July 5-9**
  - Third International Valentin Turchin Workshop on Metacomputation
    http://meta2012.pereslavl.ru/

  invited speaker
  - Neil Jones

You are invited to participate!

40 years of supercompilation (and close neighborhood)

- 1971 Yoshihiko Futamura's seminal paper
- 1972 Valentin Turchin's paper on driving
- 1974 Valentin Turchin gave lectures on supercompilation to students
- 1985 Neil Jones et al: Partial evaluation, self-application
- 1980s Valentin Turchin developed experimental supercompilers for Refal
- 1990s Valentin Turchin's supercompiler completed and improved
  - Andrei Nemytykh
- 1990s Supercompilation simplified, cross-fertilization with other methods
  - Robert Glück, Andrei Klimov, Morten Sørensen, Neil Jones
- 1999-2003 Java Supercompiler
  - Andrei Klimov, Arkady Klimov, Artem Shvorin
- 2000s Supercompilation further developed
  - Ilya Klyuchnikov, Sergei Romanenko
  - Geoff Hamilton, Neil Mitchel, Peter Jonsson, Simon Peyton Jones

Computers >1GHz, >1GB

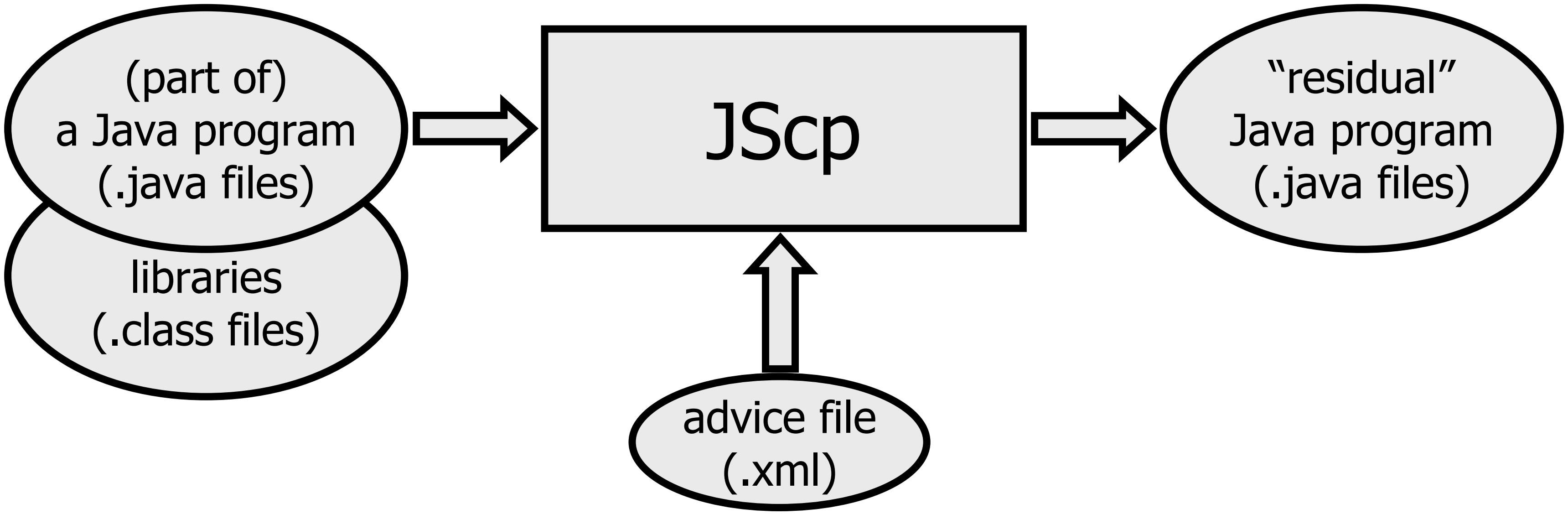
PE and SC not in practice yet!

Java Supercompiler JScp

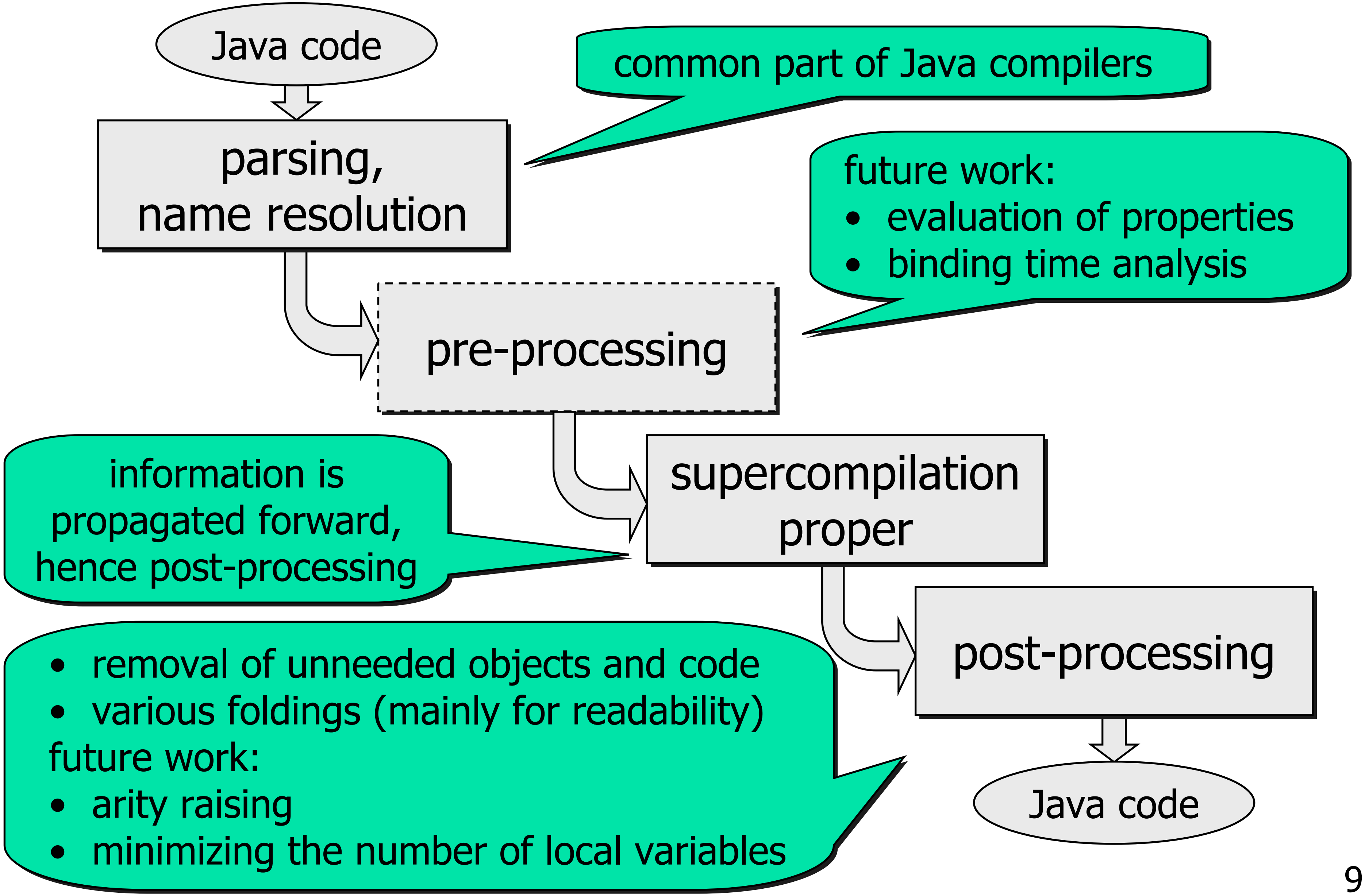
an attempt to go to practice

What is the Java Supercompiler?

JScp is a source-to-source program transformer



Java Supercompiler structure



Sample: expression interpreter

Source code

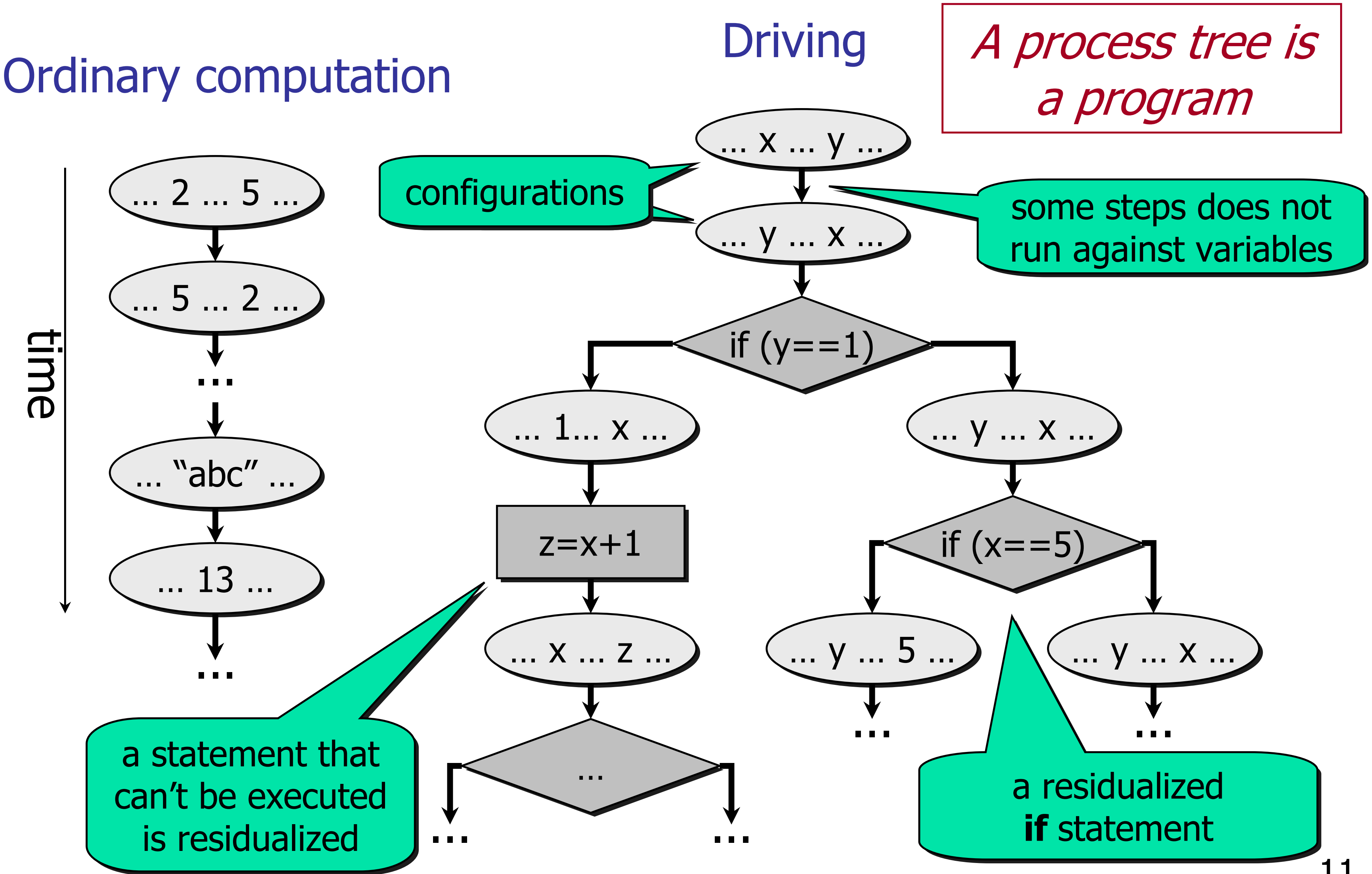
```
public static double mySqrt(double a, int iters) {
  IStatements statements =
      Assignments.create(
          new String[] { "a", "x" },  // loc var dcl
          new Assignment[] {
              new Assignment(  // x = 0.5 * (x + a/x)
              new Var("x",true),
              new Bin('*',
                  new Const(0.5),
                  new Bin('+',
                      new Var("x"),
                      new Bin('/',
                          new Var("a"),
                          new Var("x")))))
      });

  statements.setValues(new double[] {a, 1.0});
  for (int i=0; i<iters; i++) {
      statements.execute();
  }
  return statements.getValues()[1];
}
```

Residual code

```
public static double mySqrt (
  final double a_1,
  final int iters_2)
{
  final double[] values_54 = new double[2];
  values_54[0] = a_1;
  values_54[1] = 1D;
  for (int i_135 = 0; i_135 < iters_2; i_135++) {
      final double values_1_148 = values_54[1];
      values_54[1] = 0.5D *
          (values_1_148 + a_1 / values_1_148);
  }
  return values_54[1];
}
```

Driving: building process tree

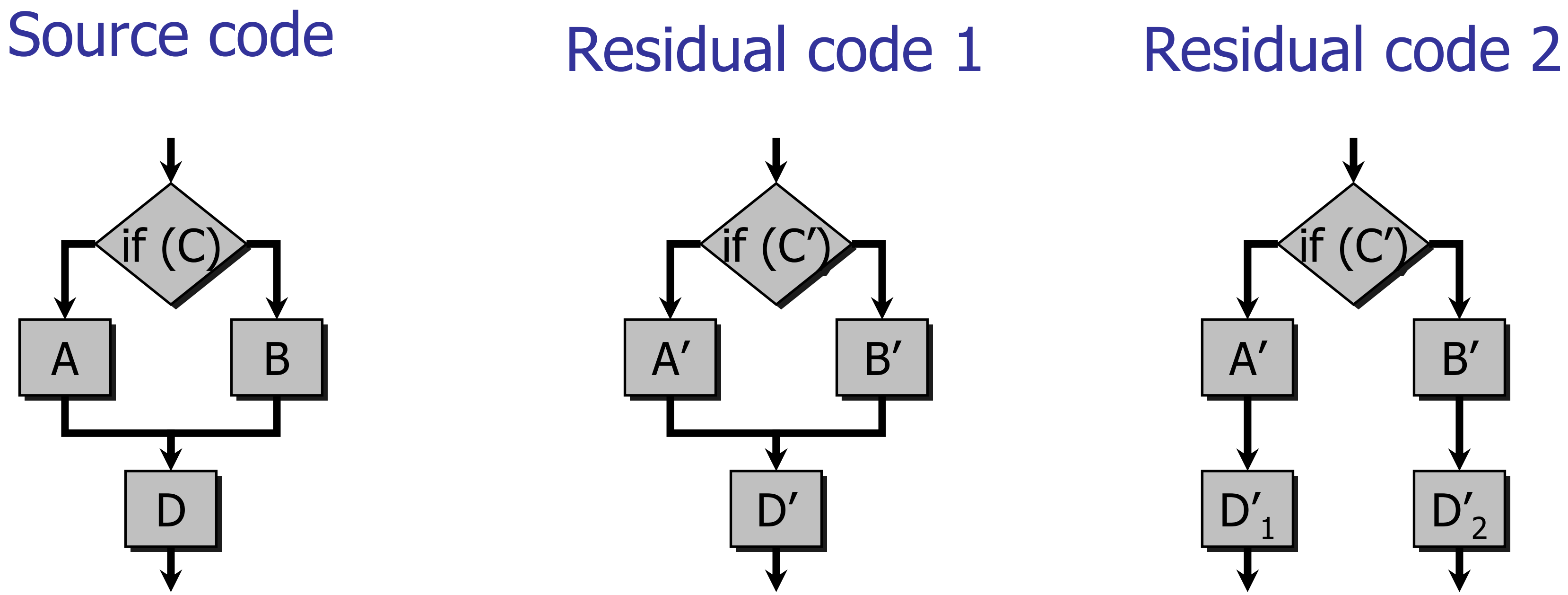


The main notions of supercompilation

- Configuration
  - a set of states = a generalized program state = a state with variables
    - a variable may occur wherever a ground value is allowed
- Driving
  - building a potentially infinite process tree
  - main problem to be solved here
    - propagation of (just enough) information about configurations
- Configuration analysis
  - folding of a process tree into finite graph
    - by reduction of a configuration to an equivalent or wider one
    - by generalization of a configuration to a wider one
    - by cutting a configuration into parts
  - main problems to be solved here
    - termination
    - choosing suitable residual program(s) among possible ones

Configuration analysis of conditional statements

- 2 alternatives to continue after statements with multiple exits

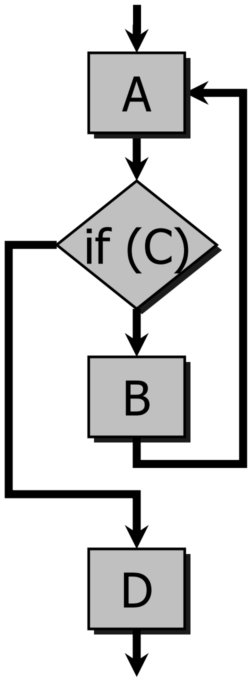


The choice is made by the human

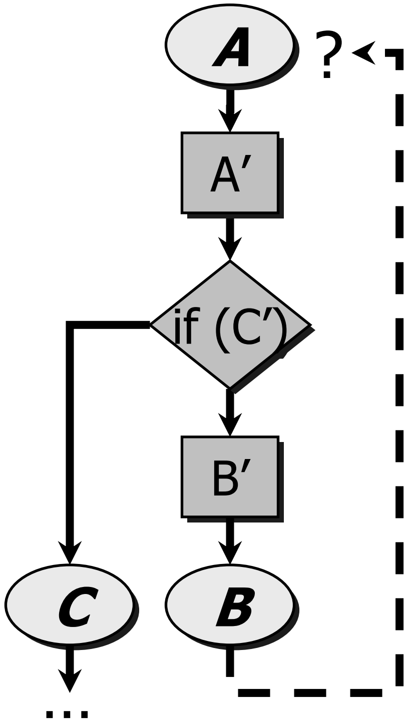
Note the possibility of exponential growth of the residual program

Configuration analysis of loops (1)

Source code



Driving…



How do configurations *A* and *B* relate?

- ***B*** ⊆ ***A*** as sets, that is
  ***B* = Δ *A*,** where **Δ** is a substitution
  then loop-back with **Δ** as an assignment

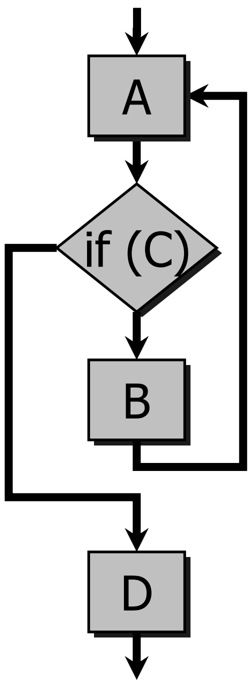
otherwise

- either
  - continue driving from ***B*** forward
- or
  - generalize ***A*** to some ***A′*** such that ***A* = Δ *A′*,** where **Δ** is a substitution
  - residualize **Δ** as assignments between configurations ***A*** and ***A′***, and
  - continue driving from ***A′***

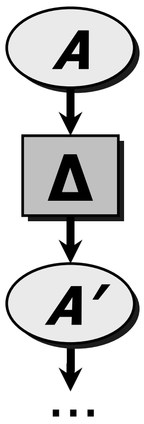
Note the possibility of exponential time to construct the residual program

Configuration analysis of loops (2)

Source code



Driving…



How do configurations *A* and *B* relate?

- ***B*** ⊆ ***A*** as sets, that is
  B* = Δ *A, where **Δ** is a substitution
  then loop-back with **Δ** as an assignment

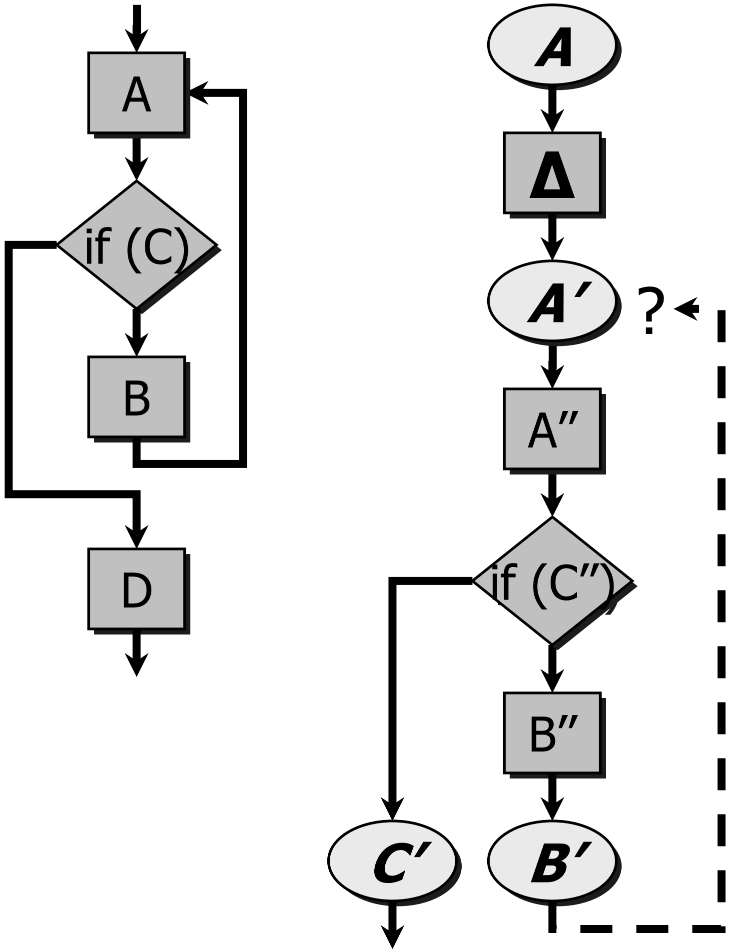
otherwise

- either
  - continue driving from ***B*** forward
- or
  - generalize ***A*** to some ***A′*** such that
    A* = Δ *A′, where **Δ** is a substitution
  - residualize **Δ** as assignments between configurations ***A*** and ***A′***, and
  - continue driving from ***A′***

Note the possibility of exponential time to construct the residual program

Configuration analysis of loops (3)

Source code Driving…



How do configurations *A* and *B* relate?

- ***B*** ⊆ ***A*** as sets, that is
  B **=** **Δ** ***A***, where **Δ** is a substitution
  then loop-back with **Δ** as an assignment

otherwise

- either
  - continue driving from ***B*** forward
- or
  - generalize ***A*** to some ***A′*** such that
    A **=** **Δ** ***A′***, where **Δ** is a substitution
  - residualize **Δ** as assignments
    between configurations ***A*** and ***A′***,
    and
  - continue driving from ***A′***

Note the possibility of exponential time to construct the residual program

Conclusions, problems and future work

Similar problems w.r.t. staging?

- Why are PE and SC not in practice yet?
- Main problem of metacomputation like partial evaluation and supercompilation
  - These are not automatic techniques like transformations in optimizing compilers
  - User control is required
  - Good human-computer interface is needed
  - Integration into studios, IDEs
- It seems small (or no) changes are required to supercompile realistic code
- Exponential blow-up can be tamed
  - The computer guarantees equivalence and presents information to the user
  - The human takes decisions where computer cannot
- Result of supercompilation:
  - Turned out to be understandable by the user (unexpectedly)
  - Studying residual graph and code helps us understand the source program
  - Debugging by analyzing residual code
  - Well-suited for further analysis and transformations, parallelization, verification
- Applications!