

Principles of Program Analysis:

A Sampler of Approaches

Transparencies based on Chapter 1 of the book: Flemming Nielson, Hanne Riis Nielson and Chris Hankin: [Principles of Program Analysis](#). Springer Verlag 2005. ©Flemming Nielson & Hanne Riis Nielson & Chris Hankin.

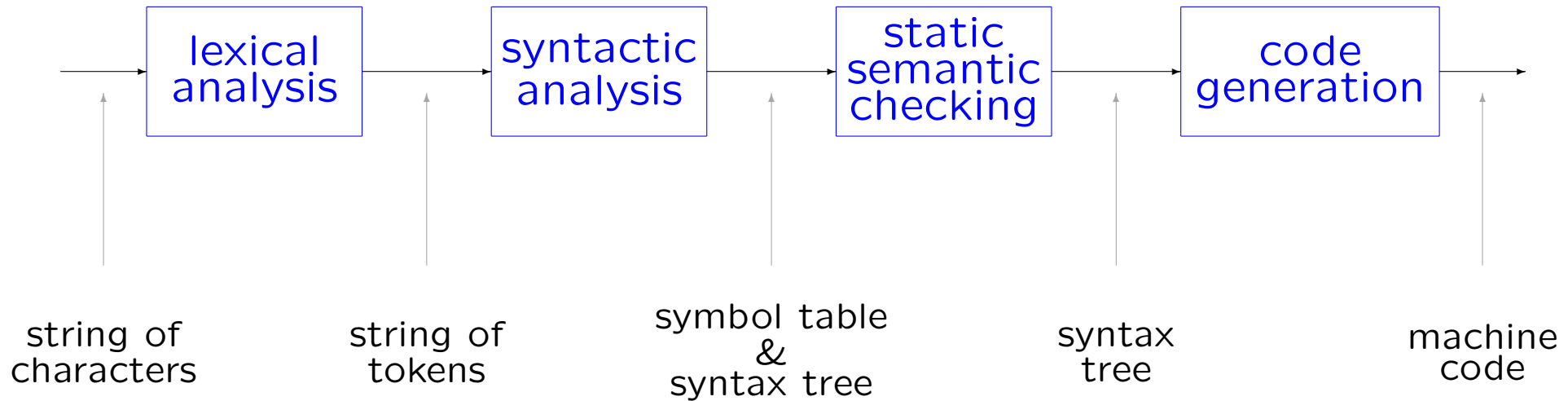
Compiler Optimisation

The classical use of program analysis is to facilitate the construction of compilers generating “optimal” code.

We begin by outlining the structure of optimising compilers.

We then prepare the setting for a worked example where we “optimise” a naive implementation of Algol-like arrays in a C-like language by performing a series of analyses and transformations.

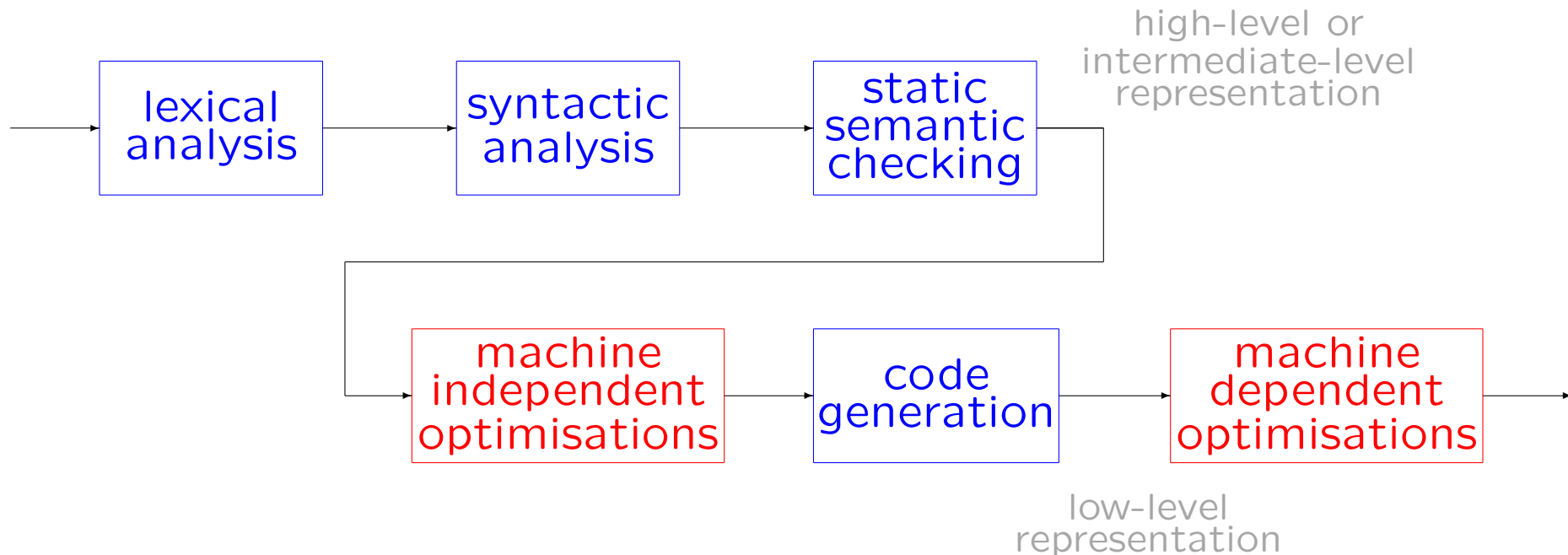
The structure of a simple compiler



Characteristics of a simple compiler:

- many phases – one or more passes
- the compiler is fast – but the code is not very efficient

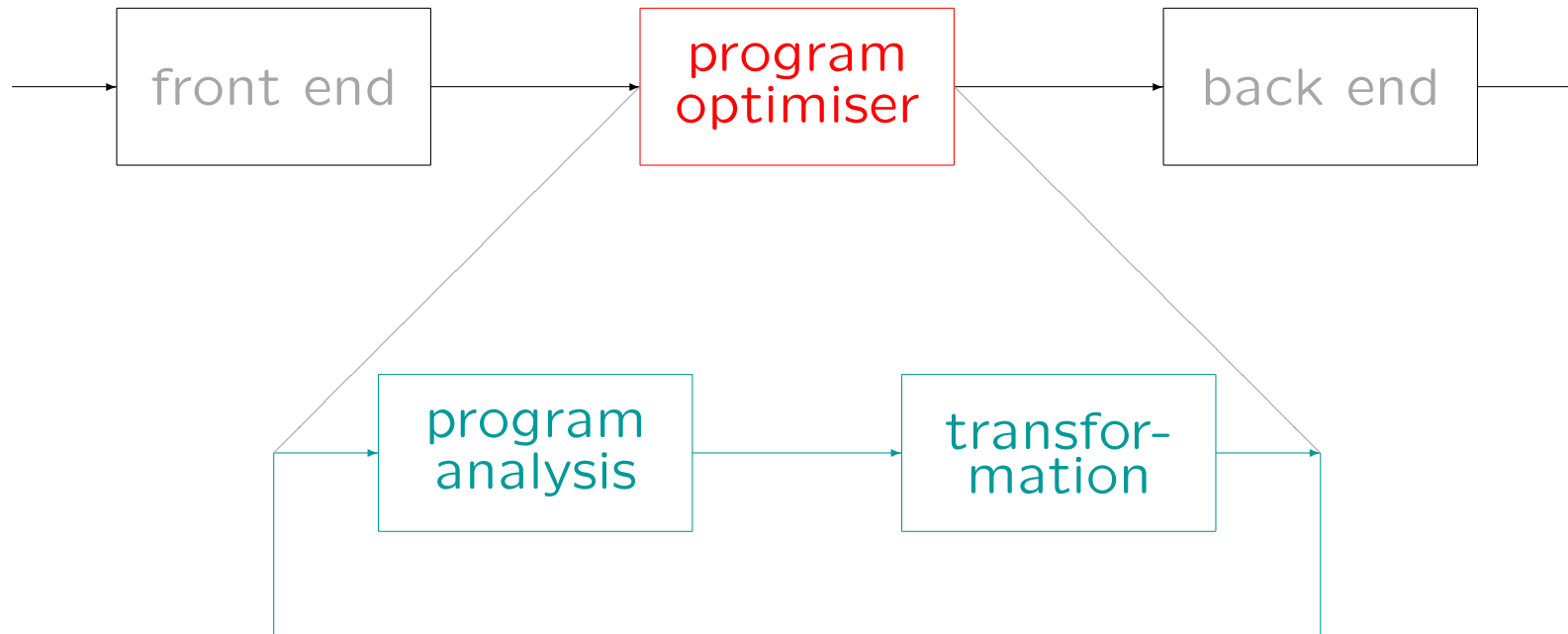
The structure of an optimising compiler



Characteristics of the optimising compiler:

- high-level optimisations: easy to adapt to new architectures
- low-level optimisations: less likely to port to new architectures

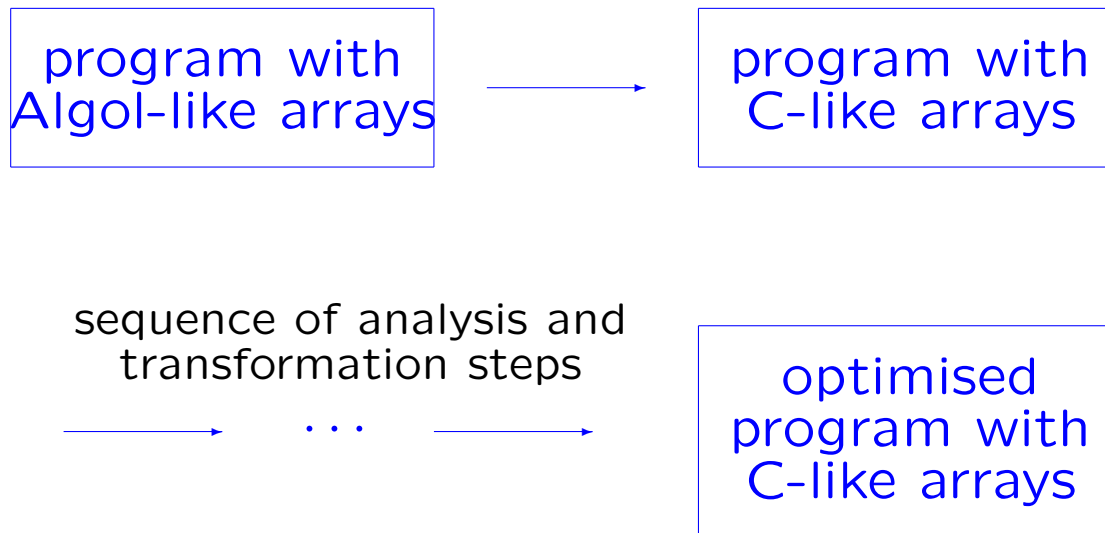
The structure of the optimisation phase



Avoid redundant computations: reuse available results, move loop invariant computations out of loops, ...

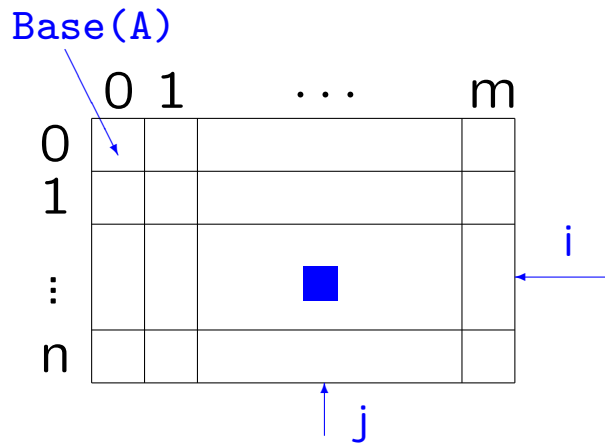
Avoid superfluous computations: results known not to be needed, results known already at compile time, ...

Example: Array Optimisation



Array representation: Algol vs. C

A: array [0:n, 0:m] of integer



Accessing the (i,j)'th element of A:

in Algol:

$A[i,j]$

in C:

$\text{Cont}(\text{Base}(A) + i * (m+1) + j)$

An example program and its naive realisation

Algol-like arrays:

```
i := 0;
while i <= n do
  j := 0;
  while j <= m do
    A[i,j] := B[i,j] + C[i,j];
    j := j+1
  od;
  i := i+1
od
```

C-like arrays:

```
i := 0;
while i <= n do
  j := 0;
  while j <= m do
    temp := Base(A) + i * (m+1) + j;
    Cont(temp) := Cont(Base(B) + i * (m+1) + j)
                + Cont(Base(C) + i * (m+1) + j);
    j := j+1
  od;
  i := i+1
od
```


Available Expressions analysis and Common Subexpression Elimination

```
i := 0;
while i <= n do
  j := 0;
  while j <= m do
    temp := Base(A) + i*(m+1) + j;
    Cont(temp) := Cont(Base(B) + i*(m+1) + j)
                 + Cont(Base(C) + i*(m+1) + j);
    j := j+1
  od;
  i := i+1
od
```

first computation

re-computations

```
t1 := i * (m+1) + j;
temp := Base(A) + t1;
Cont(temp) := Cont(Base(B)+t1)
              + Cont(Base(C)+t1);
```

Detection of Loop Invariants and Invariant Code Motion

```
i := 0;
while i <= n do
  j := 0;
  while j <= m do
    t1 := i * (m+1) + j;
    temp := Base(A) + t1;
    Cont(temp) := Cont(Base(B) + t1)
                + Cont(Base(C) + t1);
    j := j+1
  od;
  i := i+1
od
```

loop invariant

```
t2 := i * (m+1);
while j <= m do
  t1 := t2 + j;
  temp := ...
  Cont(temp) := ...
  j := ...
od
```

Detection of Induction Variables and Reduction of Strength

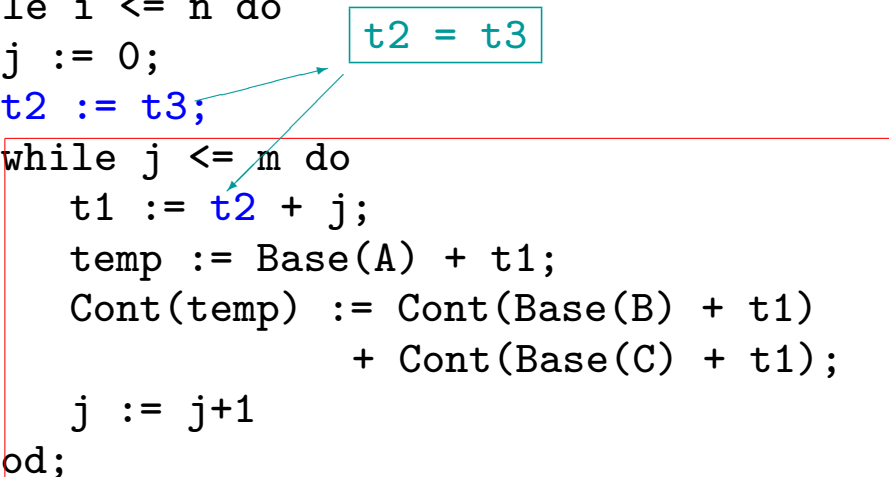
```
i := 0;
while i <= n do
  j := 0;
  t2 := i * (m+1);
  while j <= m do
    t1 := t2 + j;
    temp := Base(A) + t1;
    Cont(temp) := Cont(Base(B) + t1)
                 + Cont(Base(C) + t1);
    j := j+1
  od;
  i := i+1
od
```

induction variable

```
i := 0;
t3 := 0;
while i <= n do
  j := 0;
  t2 := t3;
  while j <= m do ... od
  i := i + 1;
  t3 := t3 + (m+1)
od
```

Equivalent Expressions analysis and Copy Propagation

```
i := 0;
t3 := 0;
while i <= n do
  j := 0;
  t2 := t3;
  while j <= m do
    t1 := t2 + j;
    temp := Base(A) + t1;
    Cont(temp) := Cont(Base(B) + t1)
                 + Cont(Base(C) + t1);
    j := j+1
  od;
  i := i+1;
  t3 := t3 + (m+1)
od
```



```
while j <= m do
  t1 := t3 + j;
  temp := ...;
  Cont(temp) := ...;
  j := ...
od
```

Live Variables analysis and Dead Code Elimination

```
i := 0;
t3 := 0;
while i <= n do
  j := 0;
  t2 := t3;
  while j <= m do
    t1 := t3 + j;
    temp := Base(A) + t1;
    Cont(temp) := Cont(Base(B) + t1)
                + Cont(Base(C) + t1);
    j := j+1
  od;
  i := i+1;
  t3 := t3 + (m+1)
od
```

dead variable

```
i := 0;
t3 := 0;
while i <= n do
  j := 0;
  while j <= m do
    t1 := t3 + j;
    temp := Base(A) + t1;
    Cont(temp) := Cont(Base(B) + t1)
                + Cont(Base(C) + t1);
    j := j+1
  od;
  i := i+1;
  t3 := t3 + (m+1)
od
```

Summary of analyses and transformations

Analysis

Transformation

Available expressions analysis

Common subexpression elimination

Detection of loop invariants

Invariant code motion

Detection of induction variables

Strength reduction

Equivalent expression analysis

Copy propagation

Live variables analysis

Dead code elimination

The Essence of Program Analysis

Program analysis offers techniques for predicting
statically at compile-time
safe & efficient **approximations**
to the set of configurations or behaviours arising
dynamically at run-time

we cannot expect
exact answers!

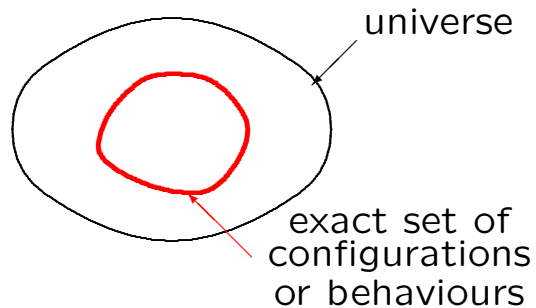
Safe: faithful to the semantics

Efficient: implementation with

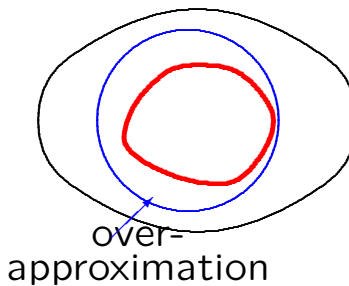
- good time performance and
- low space consumption

The Nature of Approximation

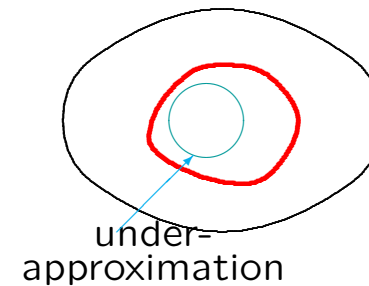
The exact world



Over-approximation



Under-approximation



Slogans: Err on the safe side!
Trade precision for efficiency!

Approaches to Program Analysis

A family of techniques ...

- data flow analysis
- constraint based analysis
- abstract interpretation
- type and effect systems
- ...
- flow logic:
a unifying framework

... that differ in their focus:

- algorithmic methods
- semantic foundations
- language paradigms
 - imperative/procedural
 - object oriented
 - logical
 - functional
 - concurrent/distributive
 - mobile
 - ...