Principles of Program Analysis:

A Sampler of Approaches

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

- **lexical analysis**
- **syntactic analysis**
- **static semantic checking**
- **code generation**

**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, ...

PPA Chapter 1 © F.Nielson & H.Riis Nielson & C.Hankin (May 2005)
Example: Array Optimisation

program with Algol-like arrays → program with C-like arrays

sequence of analysis and transformation steps → optimised program with C-like arrays
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:
  \[ \text{Cont(Base}(A) + i \times (m+1) + j) \]
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

```plaintext
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
```

```plaintext
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)
  t2 = t3
od

while j <= m do
  t1 := t3 + j;
temp := ...;
  Cont(temp) := ...;
  j := ...
  od
Live Variables analysis and Dead Code Elimination

```plaintext
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
```
## Summary of analyses and transformations

<table>
<thead>
<tr>
<th>Analysis</th>
<th>Transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available expressions analysis</td>
<td>Common subexpression elimination</td>
</tr>
<tr>
<td>Detection of loop invariants</td>
<td>Invariant code motion</td>
</tr>
<tr>
<td>Detection of induction variables</td>
<td>Strength reduction</td>
</tr>
<tr>
<td>Equivalent expression analysis</td>
<td>Copy propagation</td>
</tr>
<tr>
<td>Live variables analysis</td>
<td>Dead code elimination</td>
</tr>
</tbody>
</table>
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

**Safe:** faithful to the semantics

**Efficient:** implementation with
– good time performance and
– low space consumption

We cannot expect exact answers!
The Nature of Approximation

The exact world

Over-approximation

Under-approximation

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