CS 565 October 27, 1997
Jens Palsberg Readings: OOTS, Chapter 5

Lecture Note 10: Control-Flow Analysis

Contents

1 Type Inference

- SMALLTALK, SELF, . . . , BOPL are dynamically typed.
- Programs are concise and flexible.
- No (or only some) type annotations.
- We still want:
  - safety guarantee;
  - optimizing compiler; and
  - programming environment.
- Solution: type inference.
- This is the classical motivation.
- What we here call type inference is also known as control-flow analysis.

Type Inference in Typed Languages

- We infer types under the closed-world assumption.
- This is useful even for languages that are already typed, if they use the open-world assumption.
- Type inference gives more precise types.
- Thus, the scope of applicability includes EIFFEL, C++, BETA, and Java.
- Type inference is part of application delivery.

Key Issues

- A type is a finite set of classes.
- Each expression and variable must be given a type.
- Any type inference must be approximative.
- Too large sets are sound and conservative.
- Very large sets are useless.
- How precise information can we get?
- How fast can we compute it?

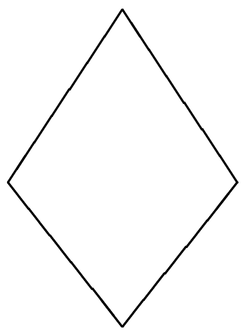
Type Constraints

- The type of an expression E is denoted by $[\![\mathrm{E}]\!]$.
- Types of different expressions satisfy mutual constraints, generated systematically from the syntax.
- The constraints are the same ones as for type checking.
- If we know all the $[\![\ldots]\!]$, then we must *verify* these constraints (type *checking*).
- If we do not know the $[\![\ldots]\!]$, then we must *solve* these constraints (type *inference*).
- If more than one solution exists, then compute a canonical one.

The Type Space

- Each type variable ranges over sets of classes, i.e., we have the lattice:

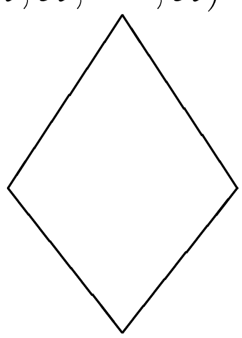
$$\mathcal{U}$$



$$\emptyset$$

- The tuple of *all* type variables ranges over the lattice:

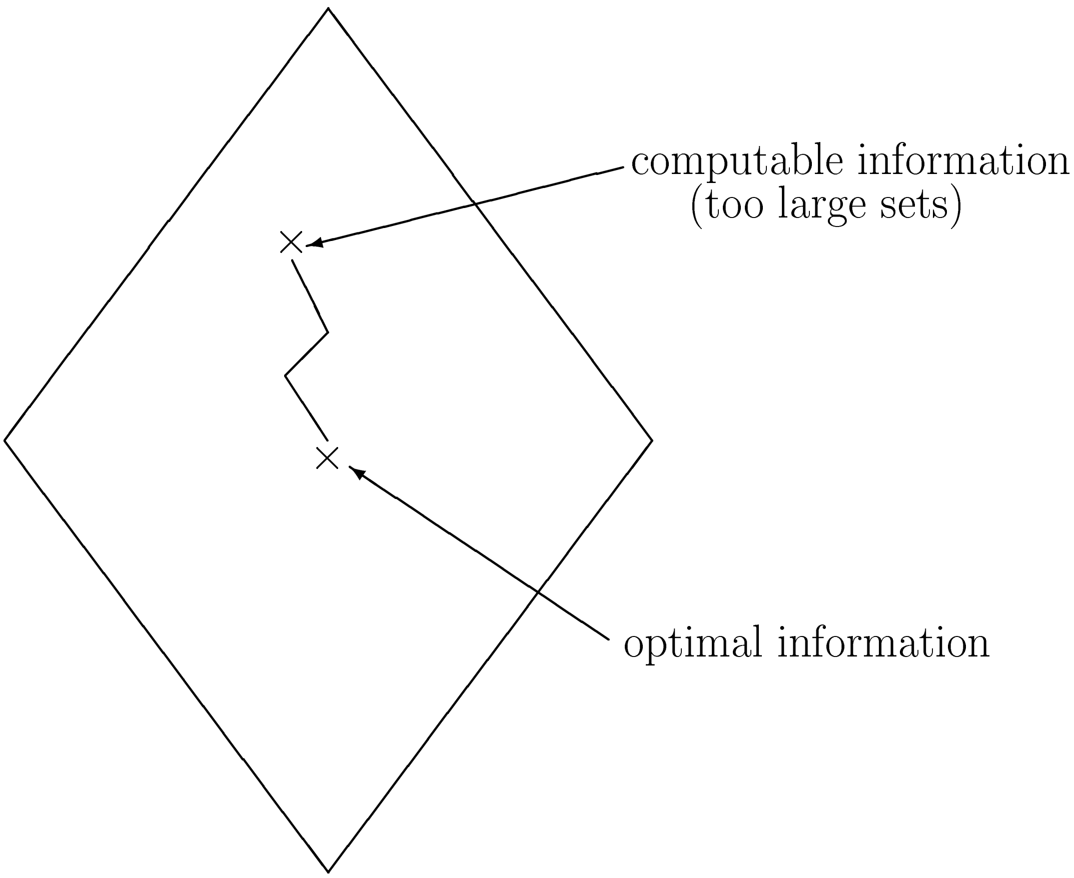
$$(\mathcal{U}, \mathcal{U}, \ldots, \mathcal{U})$$



$$(\emptyset, \emptyset, \ldots, \emptyset)$$

- Nice mathematical properties.

Optimal and Computable Information



- Computable information is above optimal information.
- The top of the lattice corresponds to trivial information.

Constraint Summary

- Types constraints are of only four kinds.
- Contributions to solutions:

$$\begin{array}{rl} C \subseteq X & \textit{start constraints} \\ X \subseteq Y & \textit{propagation constraints} \\ c \in X \Rightarrow Y \subseteq Z & \textit{conditional constraints} \end{array}$$

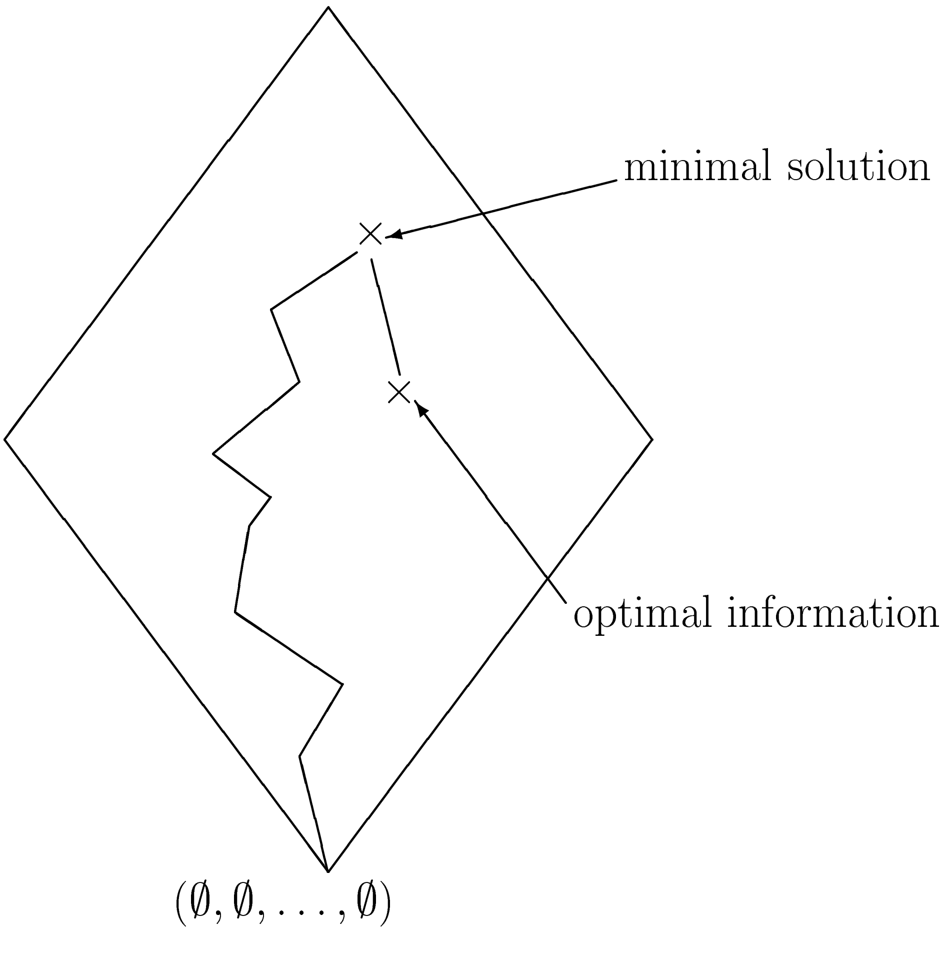
- Restrictions of solutions:

$$X \subseteq C \quad \textit{safety constraints}$$

  are ignored.
- Safety constraints can be verified *after* a solution is computed.
- Without them, there is always a unique minimal solution.

Algorithm

- The minimal solution can be computed in worst-case cubic time.
- Iterate towards the minimal solution:

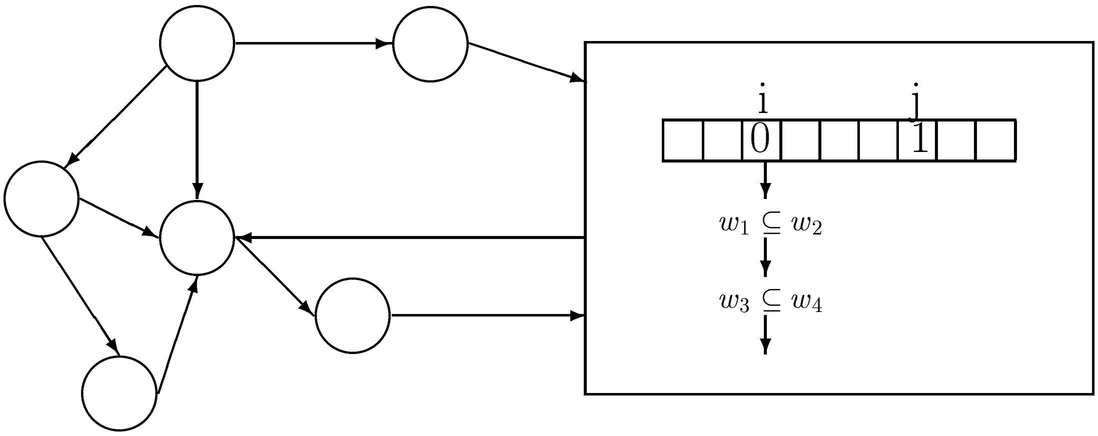


- Use clever data structures.

2 The Constraint Solver

- The central data structure.
- Accepts constraints one at the time.
- Maintains the minimal solution.
- Internal structure is a graph.
- A node for each variable.
- An edge $v \to w$ implies $v \subseteq w$.
- The value of each variable is recorded in a bit vector.
- Conditional constraints are tied to the relevant bit.

Solver Snapshot



- $B(X) = \textit{bitvector in node } X$
- $K(X, i) = \textit{list of pending constraints for bit } i \textit{ in node } X$

Solver Operations

$\text{INSERT}(C \subseteq X) =$
  for $i \in C$ **do** $\text{PROPAGATE}(X,i)$ **end**

$\text{INSERT}(X \subseteq Y) =$
  create an edge $X \to Y$;
  for $i \in B(X)$ **do** $\text{PROPAGATE}(Y,i)$ **end**

$\text{INSERT}(c \in X \Rightarrow Y \subseteq Z) =$
  if $B(X, c)$ **then**
    $\text{INSERT}(Y \subseteq Z)$
  else
    $K(X, c) := K(X, c){+}{+}(Y \subseteq Z)$
  end

$\text{PROPAGATE}(v,i) =$
  if $\neg B(v, i)$ **then**
    $B(v, i) :=$ true;
    for $v \to w$ **do** $\text{PROPAGATE}(w,i)$ **end**;
    for $k \in K(v, i)$ **do** $\text{INSERT}(k)$ **end**;
    $K(v, i) := ()$
  end

A Naïve Analysis

- Suppose the program has size n.
- There are $O(n^2)$ constraints.
- An INSERT may require $O(n)$ PROPAGATES.
- Each PROPAGATE may involve all $O(n^2)$ edges.
- Following each edge may flip $O(n)$ bits.
- Thus, worst-case time seems to be $O(n^6)$.
- But all this can never happen at once!

An Amortized Analysis

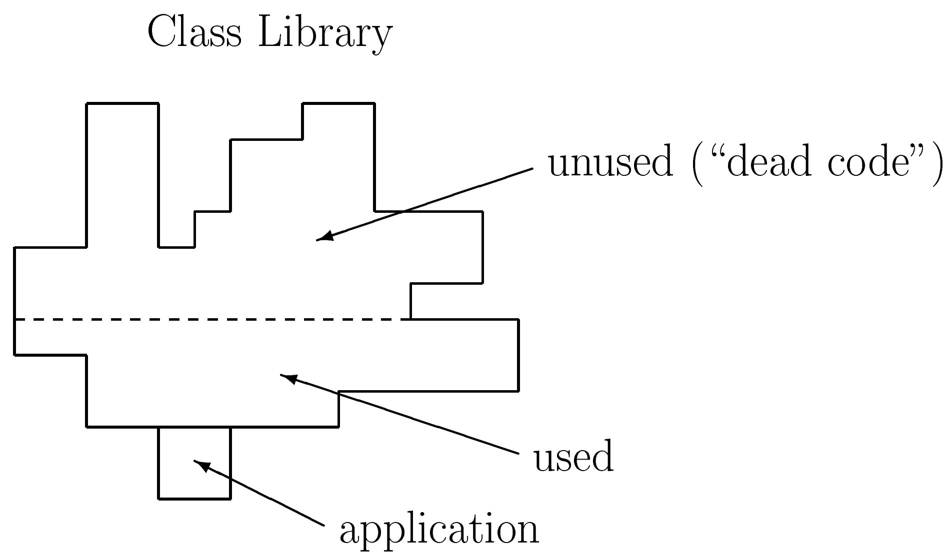
- Each activity uses up some part of the graph.
- Each bit is propagated along a specific edge at most once.
- Since there are $O(n^2)$ edges this totals $O(n^3)$.
- Each of the $O(n^2)$ constraints may:
  - be inserted into and deleted from a list once; and
  - cause the creation of a single edge,
- Which is $O(1)$ for each constraint.
- The total cost is thus $O(n^3)$.
- In practice, locality in programs gives more like linear behavior.

3 Quality of Information

- The basic algorithm is not very precise.
- It has several weak spots:
  - huge class libraries;
  - polymorphic methods; and
  - polymorphic container classes.
- They can be improved by:
  - dead code detection; and
  - code copying.
- A whole string of papers have given ever better techniques to handle these problems.
- Some techniques are general.
- Others require insight into the specific language in question.

The Trace Graph

- For a given application, most of the class library counts as "dead code".
- A trace graph gives a conservative estimate of code that can be ignored.
- This gives both more precise information and a considerable speed-up of the analysis.



The Extended Solver

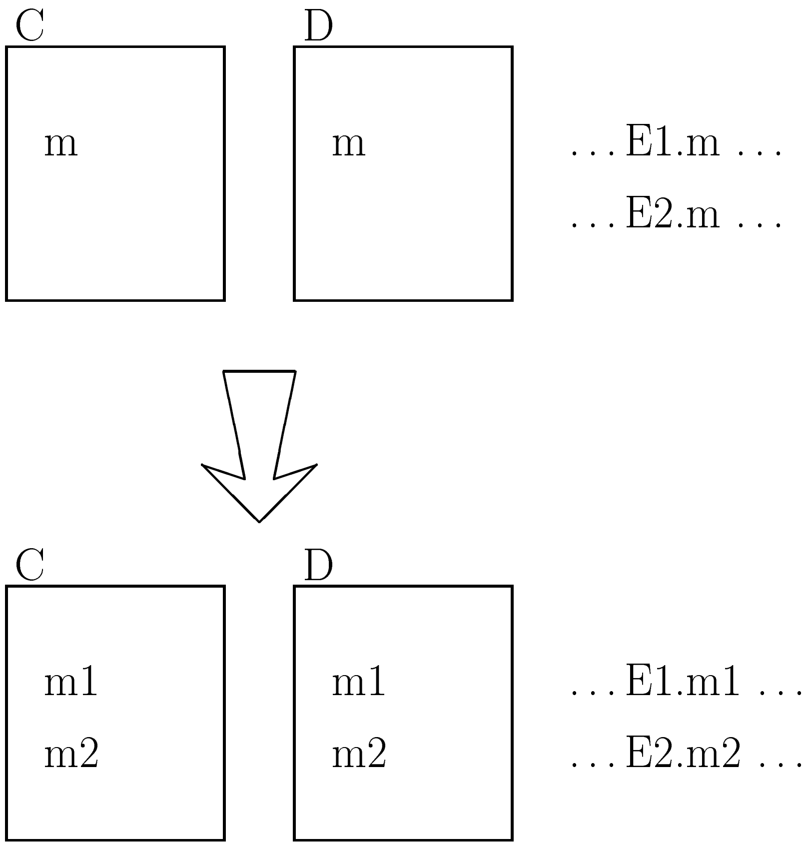
- Maintain a bit vector of *live* methods.
- Only insert constraints from the reachable parts.

```
L := ∅;
for k ∈ CONSTRAINTS(main) do
  INSERT(k)
end;
while L≠LIVE do
  choose m from LIVE\L;
  for k ∈ CONSTRAINTS(m) do
    INSERT(k);
    L := L∪{m}
  end
end;
remove all methods not in L
```

- It is still $O(n^3)$.
- The types of live methods may be improved.

Copying Methods

- A program transformation for polymorphism.
- Creates a (nominal) copy of each method implementation for each syntactic invocation.
- Can be iterated to yield ever better information.



A Copied Method

- A polymorphic use:

```
class C
  method id(x)
    x
  end
end

((C new).id(7))+10;
((C new).id(true)) or false
```

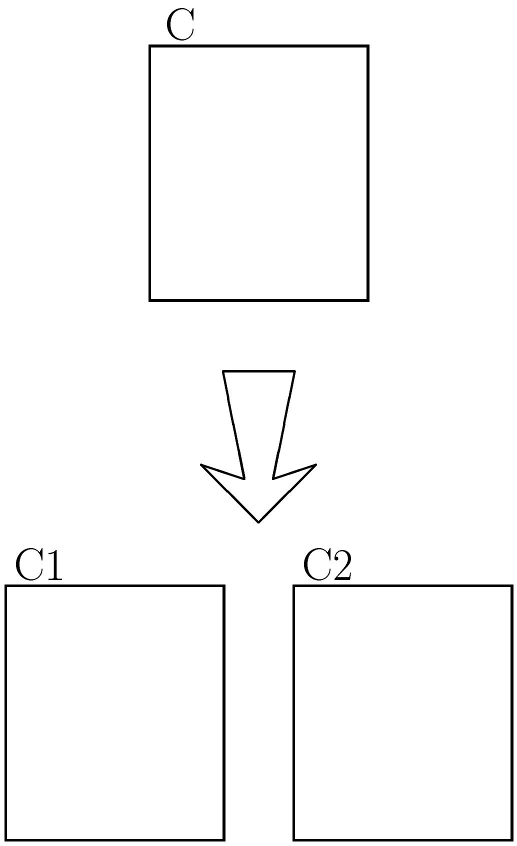
- is resolved through copying:

```
class C
  method [id@1](x)
    x
  end
  method [id@2](x)
    x
  end
end

((C new).[id@1](7))+10;
((C new).[id@2](true)) or false
```

Copying Classes

- A program transformation for container classes that gives polymorphic instance variables.
- Creates a (nominal) copy of each class for each syntactic "**new**".
- Can be iterated to yield ever better information.



…C new …

…C new …

…C1 new …

…C2 new …

A Copied Class

- A polymorphic use:

```
class Container
    var x
    method put(val)
        x := val;
        self
    end
    method get()
        x
    end
end

(Container new).put(7).get()+10;
(Container new).put(false).get() or true
```

A Copied Class (cont'd)

- is resolved through copying:

```
class [Container@1]
    var x
    method put(val)
        x := val;
        self
    end
    method get()
        x
    end
end

class [Container@2]
    ...
end

([Container@1] new).put(7).get()+10;
([Container@2] new).put(false).get() or true
```

Using Copied Code

- How does information from different copies of classes and methods relate back to the originals?
- If none of the copies require dynamic checks, then neither does the original.
- If each copy can be inlined, then the original may or may not be inlined.
- If it may not, then inline different code for each copy.
- This is *costumization*.
- For readability, an advanced browser is called for.

Strategies for Copying

- The method and class transformations are in principle sufficient.
- However, each application may square the size of the program.
- Only *adaptive* strategies will work in practice.
- Several different proposals.
- Keen understanding of the source language is required.

Summary

- Type inference by:
  - *solving* constraints.
  - fast iterative algorithm.
- Improvements by:
  - trace graph.
  - code copying.
- Widely used.
- Lifts the performance penalties of having general language constructs.
- Useful for both statically-typed and dynamically-typed languages.