Types for the Language Designer

OOPSLA'96
San Jose, California

Jens Palsberg
palsberg@cs.purdue.edu
Dept. of Computer Sciences, Purdue University
W. Lafayette, IN 47907, USA

Michael I. Schwartzbach
mis@daimi.aau.dk
BRICS
Dept. of Computer Science, University of Aarhus
Ny Munkegade, 8000 Århus C, Denmark

Biographies

- **Michael Schwartzbach** is associate professor in the BRICS Centre at the Department of Computer Science of the University of Aarhus. Michael received his Ph.D. from Cornell University in 1987. He is associate editor of the TAPOS journal.

- **Jens Palsberg** is associate professor of computer science at Purdue University. Jens received his Ph.D. from the University of Aarhus in 1992. He has been on the program committees of ECOOP'94–96.

- **Previous tutorials** at OOPSLA'92, ECOOP'93, OOPSLA'93, POPL'94, and OOPSLA'95.

- **Guest editors** of the TAPOS journal special issue on type systems (Vol. 1:3, 1995).

- **Authors** of the book Object-Oriented Type Systems (John Wiley & Sons, 1994).

Overview

1. Motivation
2. The BOPL language
3. Type Checking
4. Type Inference
5. Interfaces

Motivation

- Work in the tradition of SIMULA, BETA, C++, and EIFFEL.
- Initial observation: "types = sets of classes" gives simple description of existing type systems.
- Initial goal: to explain and improve these—not to overthrow them.
- Achievements:
  - **A common theory**, which can be used as a basis for comparisons (*this is what we originally set out to do*).
  - **Improvements** of inheritance and genericity. (*Have, so far, remained interesting theoretical comments; will not be covered today*).
  - **Type inference**, in a natural manner (*this has been picked up and used for many languages*).

The BOPL Language

BOPL

- Basic Object Programming Language.

- An untyped language model underlying our discussions.

- Supports the essential features:
  - classes and objects;
  - assignments; and
  - late binding.

- Inheritance and genericity are not considered at this stage.

The Essence of Objects

- A program consists of a *class library* and a *main expression*.
- A class contains *methods* and *instance variables*.
- An *object* is an instance of a particular class.
- The evaluation of an expression yields a value and may modify the state.
- A *value* is either an integer, a boolean, a pointer to an object, or the constant nil.

Expressions

- Object creation.
- Assignment to instance variables.
- Conditionals.
- Iteration.
- Sequencing.
- The self meta-variable.
- Message sends to objects.
- Dynamic verification of the class of an object.

Syntax

- Like SMALLTALK, but with a SIMULAsque syntax.

program	::=	classlist? exp
classlist?	::=	ε | classlist
classlist	::=	class | classlist class
class	::=	**class** id varlist? methodlist? **end** |
		class id **is** id
methodlist?	::=	ε | methodlist
methodlist	::=	method | methodlist method
method	::=	**method** id formals exp **end**
varlist?	::=	ε | varlist
varlist	::=	var | varlist var
var	::=	**var** dec
formals	::=	(declist?)
declist?	::=	ε | declist
declist	::=	dec | declist ; dec
dec	::=	idlist
idlist	::=	id | idlist , id

More Syntax

```
exp        ::=  int |
                exp binop exp |
                false |
                true |
                exp not |
                id := exp |
                exp ; exp |
                if exp then exp else exp end |
                while exp do exp end |
                nil |
                self |
                id new |
                exp class new |
                exp instance-of { idlist } |
                id |
                exp . id ( explist? ) |
                ( exp )
binop      ::=  + | - | * | = | and | or | <
explist?   ::=  ε | explist
explist    ::=  exp | explist , exp

int        ::=  digit | digit int
digit      ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
id         ::=  letter | id letter | id digit
letter     ::=  a | b | ... | z | A | B | ... | Z
```

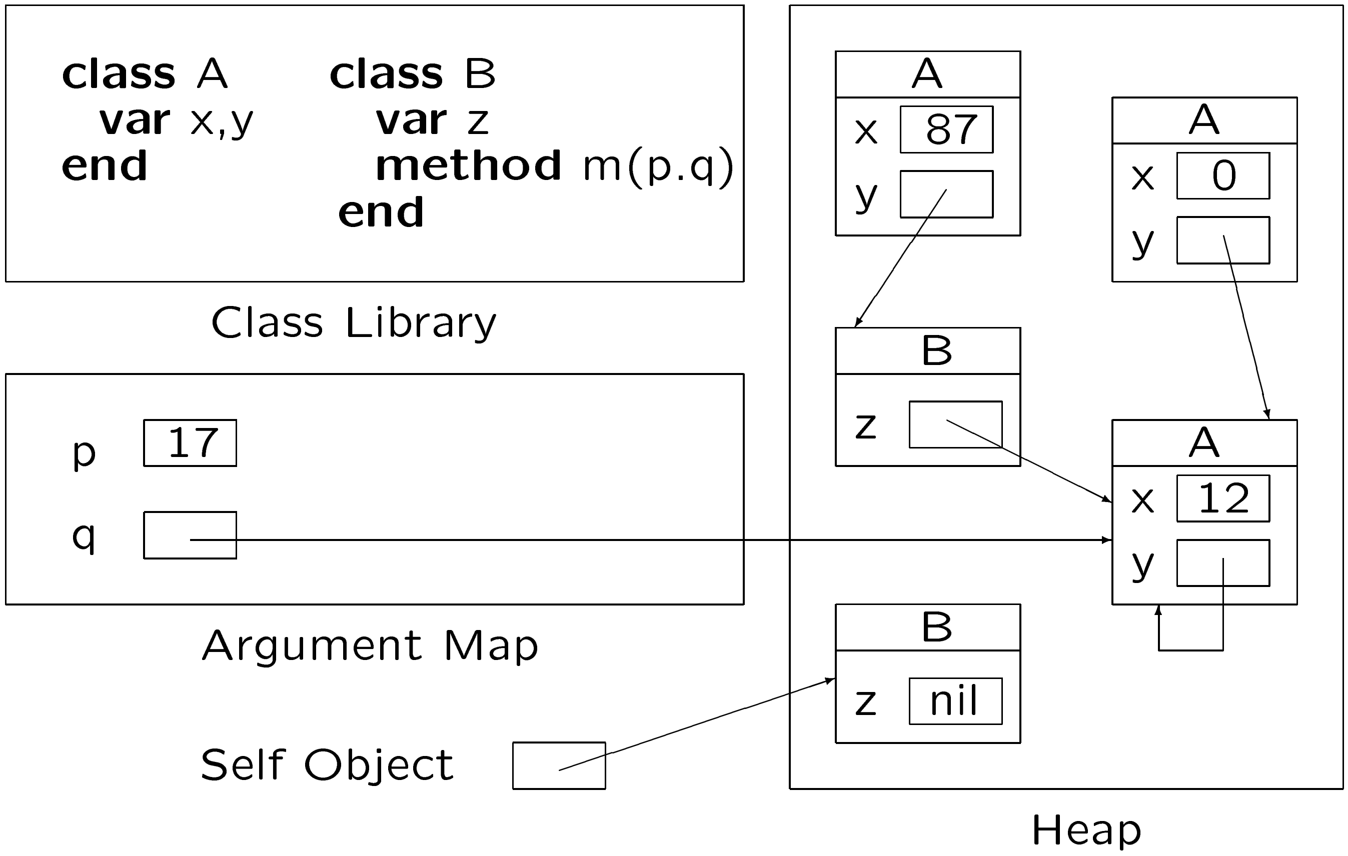
The State

- A BOPL expression is evaluated in a *state* consisting of:
  - the class library;
  - the heap;
  - the argument map; and
  - the self object.

- The *value* of an expression is either:
  - the constant nil;
  - an integer;
  - a boolean; or
  - a pointer to an object.

- An *object* is a record that contains:
  - the name of its class; and
  - a value for each instance variable.

A Possible State



- Plus run-time stack.

- Naïve implementation.

Run-Time Errors

- The possible run-time errors are:
  - type mismatch in unary and binary operators;
  - conditions in **if** and **while** not evaluating to booleans;
  - the class of a receiver not implementing a method for the message it is being sent; or
  - the receiver evaluating to nil.

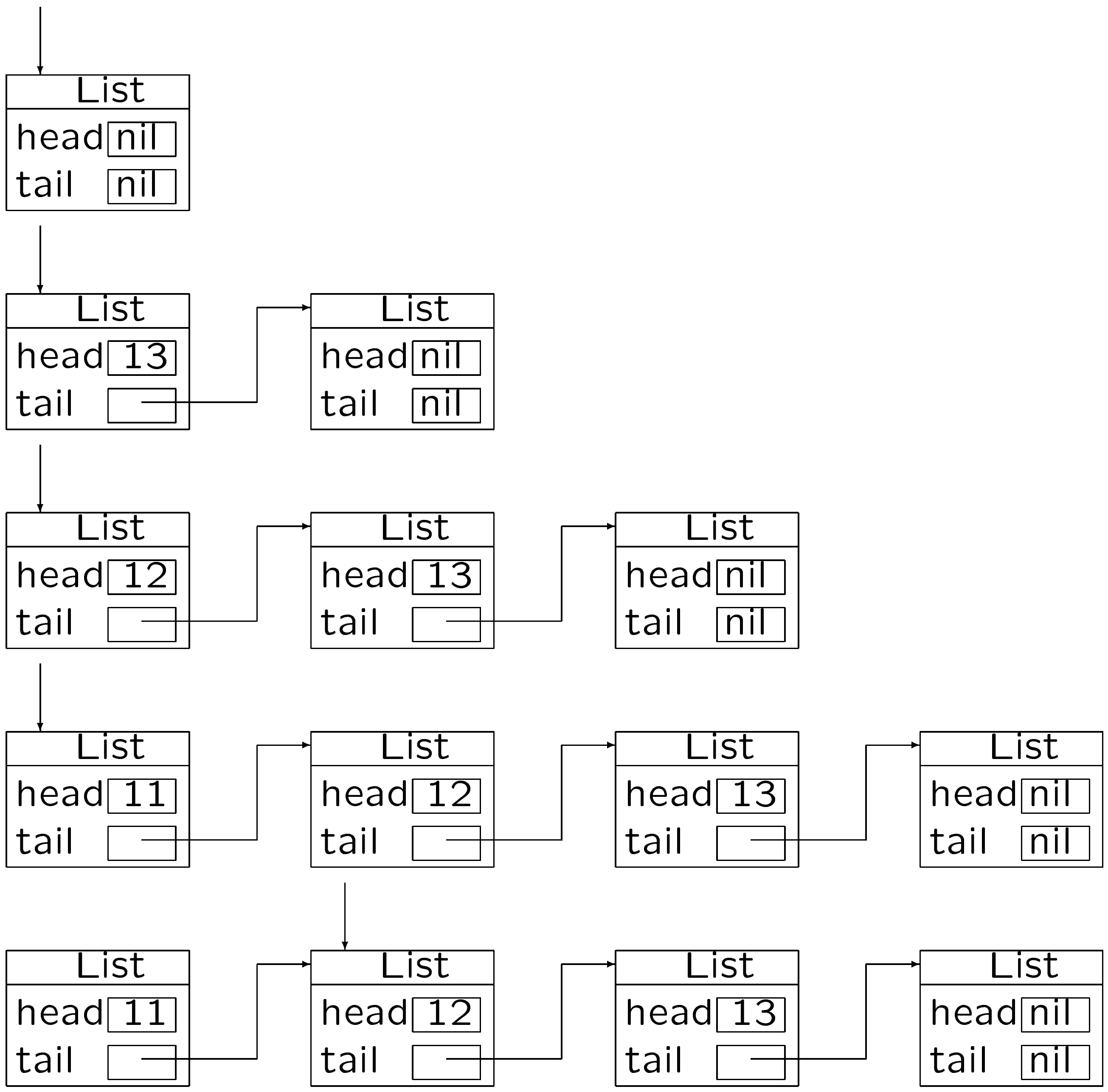
- The last two are combined known as *message-not-understood*.

Lists

```
class List
   var head, tail
   method Set(h, t)
      head := h;
      tail := t;
      self
   end
   method Cons(h)
      (self class new).Set(h, self)
   end
   method Car()
      head
   end
   method Cdr()
      tail
   end
end

(List new).Cons(13).Cons(12).Cons(11).Cdr().Car()
```

Evaluation of Lists



Arrays

```
class Array
   var head, tail, length
   method at(i)
      if (i<0) or (length<i)
      then nil
      else if i = 0 then head else tail.at(i-1) end
      end
   end
   method atput(i, x)
      if (i<0) or (length<i)
      then self
      else if i = 0
            then head := x
            else tail.atput(i-1, x)
            end
      end;
      self
   end
   method initialize(size)
      if (size<0) or (size = 0)
      then self
      else tail := (Array new).initialize(size-1);
            length := size; self
      end
   end
   method arraysize()
      length
   end
end
```

Type Checking

What is a Type?

- A type is an *invariant*.

- If a variable v contain only integer values, then we may say that v has type Int.

- Thus, the declaration

  var v: Int

  specifies an invariant about v.

- A program in which all type declarations are valid invariants is said to be *type correct*.

- This is formally undecidable, but can be approximated.

- If a type checker can verify that a program is type correct, then the program is *statically* type correct.

Why Types?

An untyped program may be:

- Unreadable.

  Types provide documentation;

  "Well-typed programs are more readable"

- Unreliable.

  Types provide a safety guarantee;

  "Well-typed programs cannot go wrong"

- Inefficient.

  Types enable optimizations;

  "Well-typed programs are faster"

World-Assumptions

- The *closed-world* assumption:

  All parts of the program are known at compile-time. Type checking is global for the entire program.

- The *open-world* assumption:

  Some parts of the program are unknown at compile-time. Type checking is local for each module.

- Both make sense, but at different stages of the development process.

- This has significant influence on typing issues.

Types are Sets of Classes

- The type of E is denoted $[\![E]\!]$.
- $[\![E]\!] = \{A, B, C, \ldots\}$.
- Invariant: E can only evaluate to nil or an instance of a class mentioned in $[\![E]\!]$.
- Generalizes the "class types" of C++, SIMULA, BETA, and Eiffel (they correspond to special sets).
- Explains subtyping, dynamic typing, and the type of nil.
- Note: only values have types, and methods are not values.
- Limitation: no type of *even* integers, unless we create a subclass.

What is Subtyping?

- When can the assignment x := E be allowed?
- The invariant on x must be respected.
- Clearly, $[\![x]\!] \supseteq [\![E]\!]$ is sufficient.
- So, subtyping is set inclusion.
- This is required of all statically type correct programs.

What is the Type of nil?

- It is $\{\}$ (the empty set).

- This satisfies the invariant:

  If nil evaluates to an instance (which it does not), then the class of that instance is in $\{\}$.

- The assignment x := nil is always allowed, since $[\![x]\!] \supseteq \{\}$.

- This becomes awkward if types are just classes: nil must then be an instance of a paradoxical common subclass of all classes.

Type Annotations

- They are placed at e.g. instance variables:

  var x: {A,B,C}

  Only instances of the classes A, B, C can get stored in the variable x.

- They have consequences throughout the program:

  $$\underset{\{A,B,C\}}{\underset{\uparrow}{E}} \;.\; m(\ldots)$$

  Control will be passed to the implementation of m in either A, B, or C.

- Uses only notions already present in the programming language.

- No elaborate language of type expressions must be mastered.

Safety Guarantee

- Avoid the error: *message-not-understood*.

- When the type of the receiver is known:

$$\underset{\{A,B,C\}}{\overset{E}{\uparrow}} \; . \; m(\,\ldots\,)$$

  check that the classes A, B, and C each implement a method named m.

- The safety of message sends at run-time is then guaranteed at compile-time.

- But the receiver may still be nil.

  (requires data flow analysis)

Optimization

- With safety guarantee, leave out checks for message-not-understood.

- Inline message sends if type of receiver is a singleton.

- Reject message sends if type of receiver is empty.

- Use hash functions to dispatch messages.

- Discard "dead code":

$$\underset{\{A,B,C\}}{\overset{}{\underset{\uparrow}{E}}} \;.\; m(\,\ldots\,)$$

  e.g. implementations of m in classes other than A, B, and C.

Constraint Technique

- Types must be *declared*, *propagated*, and *verified*.

- Constraints give a uniform presentation.

- Each expression E has a type variable $[\![E]\!]$.

- From the syntax of a program we generate constraints that express:
  - the declared type information;
  - the propagation of types; and
  - the requirements for static type correctness.

- The program is statically type correct if these constraints are solvable.

Constraints for Pascal

- Program:

```
var x,y: Integer;
var z: array of Boolean;

writeln(x+y);
z[87] := true;
```

- Type constraints:

decl	$[\![x]\!] = \text{Integer}$
decl	$[\![y]\!] = \text{Integer}$
decl	$[\![z]\!] = \textbf{array of } \text{Boolean}$
vfy	$[\![x]\!] = [\![y]\!]$
prop	$[\![x]\!] = \text{Integer} \Rightarrow [\![x+y]\!] = \text{Integer}$
prop	$[\![x]\!] = \text{Real} \Rightarrow [\![x+y]\!] = \text{Real}$
prop	$[\![87]\!] = \text{Integer}$
prop	$[\![\text{true}]\!] = \text{Boolean}$
vfy	$[\![z[87]]\!] = [\![\text{true}]\!]$
prop	$[\![z]\!] = \textbf{array of } [\![z[87]]\!]$

Annotations in BOPL

- For now, types are finite sets of classes.

- We will attach types three places in programs.

- At instance variables; the associated invariant is that the instance variable can only contain nil or the values described by the type.

- At formal arguments; the associated invariant is the same as for instance variables.

- At method results; the associated invariant is that the body of the method can only evaluate to nil or the vales described by the type.

An Annotated Program

```
class A
   var x: A
   var y: {A,B,Int}
   method m(f: B) returns B
      f.m(x)
   end
end

class B
   method m(g: A) returns B
      self
   end
end

(A new).m(B new).m(nil)
```

Constraints for BOPL

- Type variables as before.

- Assume that all variable and argument names are distinct (rename if necessary).

- Use the notation:

  $$[\![x]\!],\ [\![f]\!],\ \text{and}\ [\![E]\!]$$

  for the type variables associated with the instance variable x, the formal argument f, and the expression E.

- Use the notation $[\![\text{self-C}]\!]$ for the self meta-variable in the class C.

- Generate constraints from syntax of expressions.

Type Constraints

1) "int". For any integer constant k we have:

$$[\![k]\!] = \{\text{Int}\}$$

2) "true" and "false". For the boolean constants we have:

$$[\![\text{true}]\!] = \{\text{Bool}\} \quad \text{and} \quad [\![\text{false}]\!] = \{\text{Bool}\}$$

3) "nil". For the nil constant we have the constraint:

$$[\![\text{nil}]\!] = \{\}$$

4) "exp_1 + exp_2". For the addition we have:

$$\begin{gathered} [\![\text{exp}_1]\!] = \{\text{Int}\} \\ [\![\text{exp}_2]\!] = \{\text{Int}\} \\ [\![\text{exp}_1 + \text{exp}_2]\!] = \{\text{Int}\} \end{gathered}$$

11) "exp not". For the negation we have:

$$\begin{gathered} [\![\text{exp}]\!] = \{\text{Bool}\} \\ [\![\text{exp not}]\!] = \{\text{Bool}\} \end{gathered}$$

Type Constraints

12) "id := exp". For the assignment we have:

$$[\![\text{exp}]\!] \subseteq [\![\text{id}]\!]$$
$$[\![\text{id} := \text{exp}]\!] = [\![\text{exp}]\!]$$

13) "exp_1 ; exp_2". For the sequence we have:

$$[\![\text{exp}_1 \ ; \ \text{exp}_2]\!] = [\![\text{exp}_2]\!]$$

14) "**if** exp_1 **then** exp_2 **else** exp_3 **end**". For the conditional we have:

$$[\![\text{exp}_1]\!] = \{\text{Bool}\}$$
$$[\![\textbf{if} \ \text{exp}_1 \ \textbf{then} \ \text{exp}_2 \ \textbf{else} \ \text{exp}_3 \ \textbf{end}]\!]$$
$$= [\![\text{exp}_2]\!] \cup [\![\text{exp}_3]\!]$$

15) "**while** exp_1 **do** exp_2 **end**". For the iteration we have:

$$[\![\text{exp}_1]\!] = \{\text{Bool}\}$$
$$[\![\textbf{while} \ \text{exp}_1 \ \textbf{do} \ \text{exp}_2 \ \textbf{end}]\!] = \{\}$$

Type Constraints

16) "self". For the self meta-variable we have:

$$[\![\text{self-C}]\!] = \{\text{C}\}$$

17) "id **new**". For object creation we have

$$[\![\text{id } \textbf{new}]\!] = \{\text{id}\}$$

18) "exp **class new**". For indirect object creation we have:

$$[\![\text{exp}]\!] \subseteq \{\textit{all classes}\}$$
$$[\![\text{exp } \textbf{class new}]\!] = [\![\text{exp}]\!]$$

19) "exp **instance-of** {idlist}". For dynamic check of class membership we have:

$$[\![\text{exp } \textbf{instance-of} \ \{ \text{ idlist } \}]\!] = \{ \text{ idlist } \}$$

20) "id". For a variable or a formal argument we have:

$$[\![\text{id}]\!] = \textit{the declared type}$$

Type Constraints

21) "exp . id (exp_1, . . . , exp_n)". For the message send, the constraints are:

$$[\![\text{exp}]\!] \subseteq \{\textit{all legal receiver classes}\}$$

$$\text{C} \in [\![\text{exp}]\!] \Rightarrow \begin{cases} [\![\text{exp}_1]\!] \subseteq [\![\text{id}_1]\!] \\ \vdots \\ [\![\text{exp}_n]\!] \subseteq [\![\text{id}_n]\!] \\ [\![\text{exp.id(exp}_1\text{,...,exp}_n\text{)}]\!] \supseteq \text{type} \supseteq [\![\text{exp}_0]\!] \end{cases}$$

where C implements a method for the message:

method id(id_1:type_1, . . . , id_n:type_n) **returns** type
  exp_0
end

An Example

- The program:

```
class A
    var x: A
    var y: {A,B,Int}
    method m(f: B) returns B
        f.m(x)
    end
end

class B
    method m(g: A) returns B
        self
    end
end

(A new).m(B new).m(nil)
```

- Types for all subexpressions:

$\llbracket \mathrm{x} \rrbracket = \{\mathrm{A}\}$
$\llbracket \mathrm{y} \rrbracket = \{\mathrm{A,B,Int}\}$
$\llbracket \mathrm{f} \rrbracket = \{\mathrm{B}\}$
$\llbracket \mathrm{g} \rrbracket = \{\mathrm{A}\}$
$\llbracket \mathrm{f.m(x)} \rrbracket = \{\mathrm{B}\}$
$\llbracket \mathrm{self\text{-}B} \rrbracket = \{\mathrm{B}\}$
$\llbracket \mathrm{A}\ \mathbf{new} \rrbracket = \{\mathrm{A}\}$
$\llbracket \mathrm{B}\ \mathbf{new} \rrbracket = \{\mathrm{B}\}$
$\llbracket \mathrm{(A}\ \mathbf{new}\mathrm{).m(B}\ \mathbf{new}\mathrm{)} \rrbracket = \{\mathrm{B}\}$
$\llbracket \mathrm{nil} \rrbracket = \{\}$
$\llbracket \mathrm{(A}\ \mathbf{new}\mathrm{).m(B}\ \mathbf{new}\mathrm{).m(nil)} \rrbracket = \{\mathrm{B}\}$

More Example

The constraints:

2) 20) $[\![x]\!] = \{A\}$
3) 20) $[\![y]\!] = \{A,B,Int\}$
4) 20) $[\![f]\!] = \{B\}$
10) 20) $[\![g]\!] = \{A\}$
5) 21) $[\![f]\!] \subseteq \{A,B\}$
5) 22) $A \in [\![f]\!] \Rightarrow \begin{cases} [\![x]\!] \subseteq [\![f]\!] \\ [\![f.m(x)]\!] \supseteq \{B\} \supseteq [\![f.m(x)]\!] \end{cases}$
5) 22) $B \in [\![f]\!] \Rightarrow \begin{cases} [\![x]\!] \subseteq [\![g]\!] \\ [\![f.m(x)]\!] \supseteq \{B\} \supseteq [\![\text{self-B}]\!] \end{cases}$
11) 16) $[\![\text{self-B}]\!] = \{B\}$
15) 17) $[\![A\ \mathbf{new}]\!] = \{A\}$
15) 17) $[\![B\ \mathbf{new}]\!] = \{B\}$
15) 21) $[\![A\ \mathbf{new}]\!] \subseteq \{A,B\}$
15) 22) $A \in [\![A\ \mathbf{new}]\!] \Rightarrow$
$\begin{cases} [\![B\ \mathbf{new}]\!] \subseteq [\![f]\!] \\ [\![(A\ \mathbf{new}).m(B\ \mathbf{new})]\!] \supseteq \{B\} \supseteq [\![f.m(x)]\!] \end{cases}$
15) 22) $B \in [\![A\ \mathbf{new}]\!] \Rightarrow$
$\begin{cases} [\![B\ \mathbf{new}]\!] \subseteq [\![g]\!] \\ [\![(A\ \mathbf{new}).m(B\ \mathbf{new})]\!] \supseteq \{B\} \supseteq [\![\text{self-B}]\!] \end{cases}$
15) 3) $[\![nil]\!] = \{\}$
15) 21) $[\![(A\ \mathbf{new}).m(B\ \mathbf{new})]\!] \subseteq \{A,B\}$
15) 22) $A \in [\![(A\ \mathbf{new}).m(B\ \mathbf{new})]\!] \Rightarrow$
$\begin{cases} [\![nil]\!] \subseteq [\![f]\!] \\ [\![(A\ \mathbf{new}).m(B\ \mathbf{new}).m(nil)]\!] \supseteq \{B\} \supseteq [\![f.m(x)]\!] \end{cases}$
15) 22) $B \in [\![(A\ \mathbf{new}).m(B\ \mathbf{new})]\!] \Rightarrow$
$\begin{cases} [\![nil]\!] \subseteq [\![g]\!] \\ [\![(A\ \mathbf{new}).m(B\ \mathbf{new}).m(nil)]\!] \supseteq \{B\} \supseteq [\![\text{self-B}]\!] \end{cases}$

Dynamic Type Checks

- ```
  method TransPrint(g: {Circle,Rectangle,Triangle})
                       returns {Circle,Rectangle,Triangle}
    g.Translate(100,200);
    g.Print;
    g
  end

  c:=(App new).TransPrint(c)
  ```

- Not statically correct, but yields no run-time error.

- The compiler responds:

  – *I know what I am doing. Go away!*
    The program has been rejected for a good reason.

  – *This is not my problem, its yours!*
    The programmer may insert a dynamic check.

  – *I will take care of this!*
    The compiler will insert a dynamic check.

- c:=((App **new**).TransPrint(c) **instance-of** Circle)
  ```

When to Use Them

- If dynamic checks are inserted blindly, then any program will be accepted by the compiler.

- Want to make use of the declared type information as far as possible.

1) When a program fragment is statically correct, then it is accepted and no dynamic checks are inserted.

2) When a program fragment is statically incorrect but might succeed at run-time, then it is accepted and dynamic checks are inserted.

3) When a program fragment is statically incorrect and cannot succeed at run-time, then it is rejected.

- Minimize the number of dynamic checks and filter out most meaningless programs.

How to Use Them

- At an assignment id := exp.

- Constraint is $[\![\text{id}]\!] \supseteq [\![\text{exp}]\!]$.

1) $[\![\text{id}]\!] \supseteq [\![\text{exp}]\!]$. The expression will always evaluate to an object of a legal class.

2) $[\![\text{id}]\!] \cap [\![\text{exp}]\!] \neq \{\}$. The expression might evaluate to an object of a legal class. The program is accepted after the assignment has been transformed into id := (exp **instance-of** $[\![\text{id}]\!]$).

3) $[\![\text{id}]\!] \cap [\![\text{exp}]\!] = \{\}$. The expression cannot possibly evaluate to an object of a legal class. The program is rejected.

- Similarly for arguments and return values.

An Example

- Before:

```
class A
    var x: {Int, Bool}
    var y: Int
    method m(f: A) returns Int
        y := x
    end
end

class B
    method m(g: {A, B}) returns Int
        g.m(g)
    end
end
```

- After:

```
class A
    var x: {Int, Bool}
    var y: Int
    method m(f: A) returns Int
        y := (x instance-of Int)
    end
end

class B
    method m(g: {A, B}) returns Int
        g.m(g instance-of A)
    end
end
```

Inheritance: Class Types

- type ::= Void | Int | Bool | $\uparrow$C

- Special sets:

  – the expression Void specifies the *empty set*;
  – the expressions Int and Bool specify the sets {Int} and {Bool}; and
  – the expression $\uparrow$C specifies the set of C and all its subclasses.

- Under the closed-world assumption, they denote finite sets.

- Under the open-world assumption, they denote infinite sets.

- Type constraints can be systematically derived from the general ones.

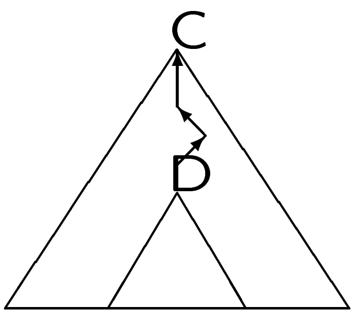
Properties

- Subset inclusion can be decided:

$\subseteq$	Void	Int	Bool	$\uparrow$C
Void	*true*	*true*	*true*	*true*
Int	*false*	*true*	*false*	*false*
Bool	*false*	*false*	*true*	*false*
$\uparrow$D	*false*	*false*	*false*	D $\leq$ C

- D $\leq$ C is transitive, reflexive subclassing.

- For class types, *subtyping* is *subclassing*.

- 

- Class types are closed under intersection, but *not* under union.

Derived Type Constraints

1) "int". For any integer constant k we have:

$$[\![k]\!] = \mathrm{Int}$$

2) "true" and "false". We have:

$$[\![\mathrm{true}]\!] = \mathrm{Bool} \quad \text{and} \quad [\![\mathrm{false}]\!] = \mathrm{Bool}$$

3) "nil". For the nil constant we have:

$$[\![\mathrm{nil}]\!] = \mathrm{Void}$$

4) "exp_1 + exp_2". For the addition we have:

$$\begin{array}{l} [\![\mathrm{exp}_1]\!] = \mathrm{Int} \wedge [\![\mathrm{exp}_2]\!] = \mathrm{Int} \\ [\![\mathrm{exp}_1 + \mathrm{exp}_2]\!] = \mathrm{Int} \end{array}$$

12) "id := exp". For the assignment we have:

$$[\![\mathrm{exp}]\!] \subseteq [\![\mathrm{id}]\!] \wedge [\![\mathrm{id} := \mathrm{exp}]\!] = [\![\mathrm{exp}]\!]$$

Derived Type Constraints

14) "**if** exp_1 **then** exp_2 **else** exp_3 **end**". We have:

$$[\![\text{exp}_1]\!] = \text{Bool}$$
$$[\![\textbf{if } \text{exp}_1 \textbf{ then } \text{exp}_2 \textbf{ else } \text{exp}_3 \textbf{ end}]\!] = [\![\text{exp}_2]\!] = [\![\text{exp}_3]\!]$$

19) "exp **instance-of** $\Uparrow$id". For the dynamic check of class membership we have:

$$[\![\text{exp } \textbf{instance-of } \Uparrow\text{id}]\!] = \Uparrow\text{id}$$

21) Message sends:

$$[\![\text{exp}]\!] \subseteq \{\textit{all legal receiver classes}\}$$

This constraint is false if $[\![\text{exp}]\!]$ is Int or Bool. If $[\![\text{exp}]\!] = \Uparrow\text{C}$ then C must implement a method for the message.

$$\text{C} \in [\![\text{exp}]\!] \Rightarrow \ldots$$

Suppose $[\![\text{exp}]\!]$ is a cone set of the form $\Uparrow$D for some class D. The condition $\text{C} \in [\![\text{exp}]\!]$ can then be verified by checking that C is a subclass of D.

No Surprises

- The derived constraints correspond closely to the type rules for C++, SIMULA, BETA, and EIFFEL.

- Supports the open-world assumption and, hence, separate compilation.

- Gives quite a few more dynamic checks, since the types are much coarser.

Other Type Systems

- A general recipe for new type systems.
- Define new type expressions.
- Show how to expand into sets.
- Under the closed-world assumption, that's it!
- Under the open-world assumption, specialize the type constraints.
- This may involve development of new algorithms for $\subseteq$, $\cap$, and $\cup$ on type expressions.
- Suggestions: union types, interfaces.

Summary

- Types improve the readability, reliability, and efficiency of programs.
- A type is a set of classes, and subtyping is set inclusion.
- A type expresses an invariant of run-time behavior.
- The programs that satisfy this invariant are called type correct.
- Static type correctness is a decidable property that implies type correctness.
- Dynamic type correctness holds when the type invariants are checked at run-time.
- Dynamic typing is necessary for realistic applications.

Type Inference

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.

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++, and BETA.

- 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 $[\![E]\!]$.

- Types of different expressions satisfy mutual constraints, generated systematically from the syntax.

- The constraints are the same ones as for type checking.

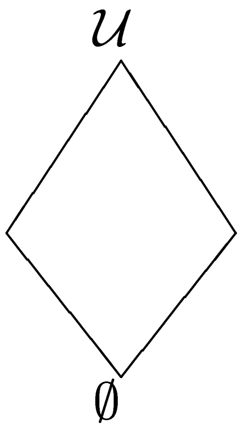
- If we know all the $[\![\ldots]\!]$, then we must *verify* these constraints (type *checking*).

- If we do not know the $[\![\ldots]\!]$, then we must *solve* these constraints (type *inference*).

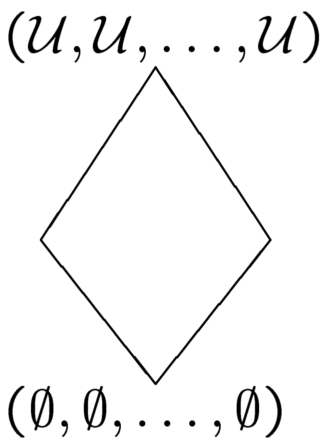
- If more than one solution exists, then compute a canonical one.

The Type Space

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

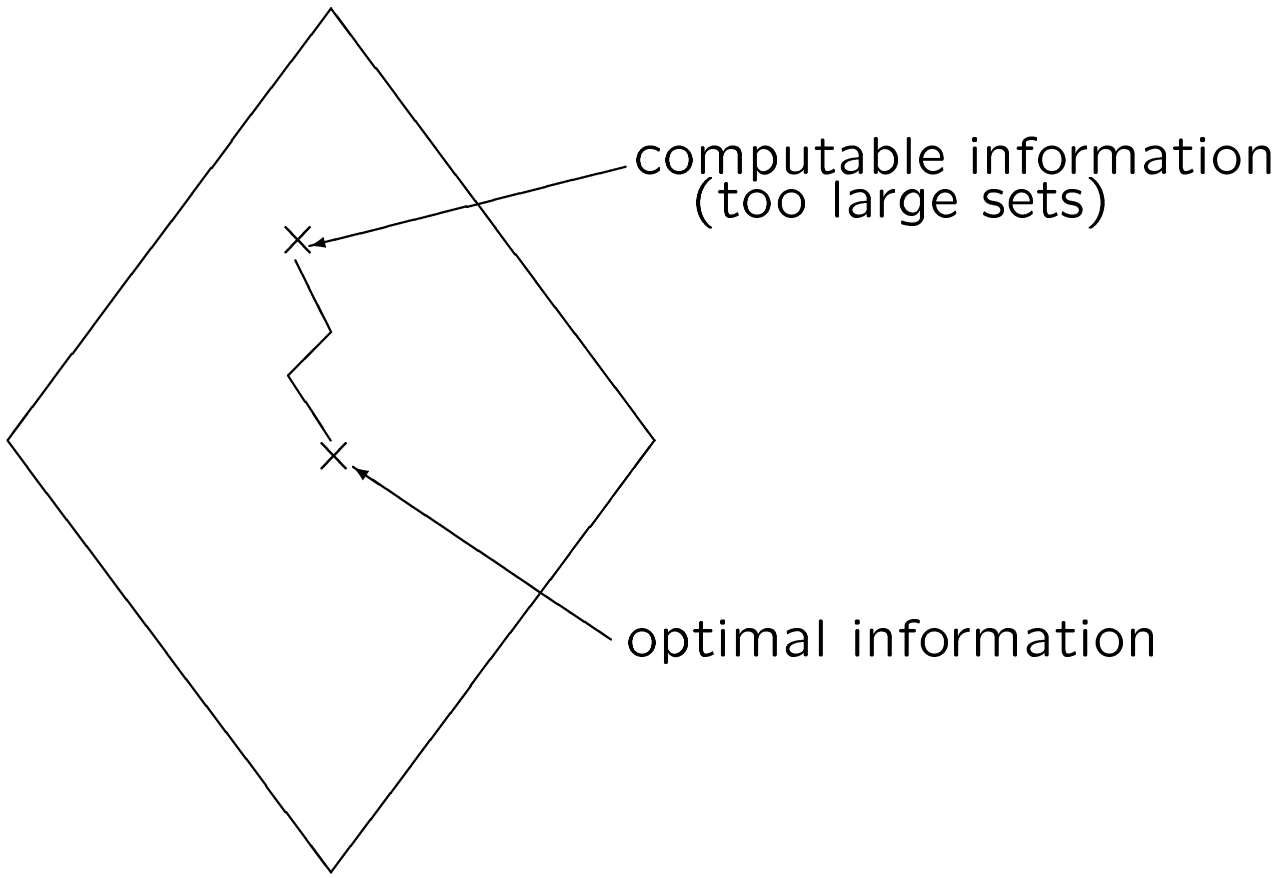


- The tuple of *all* type variables ranges over the lattice:



- 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.

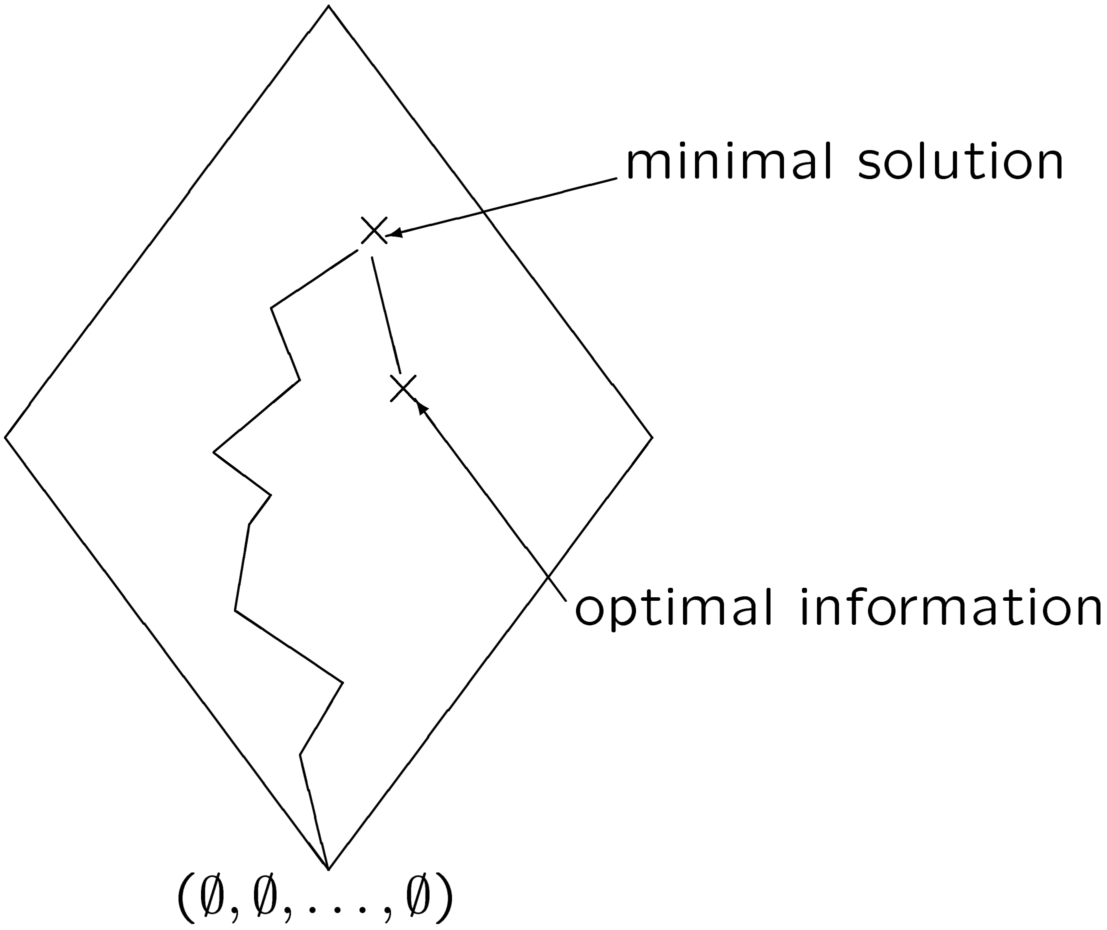
- Safety constraints can be verified *after* solution is computed.

- 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:

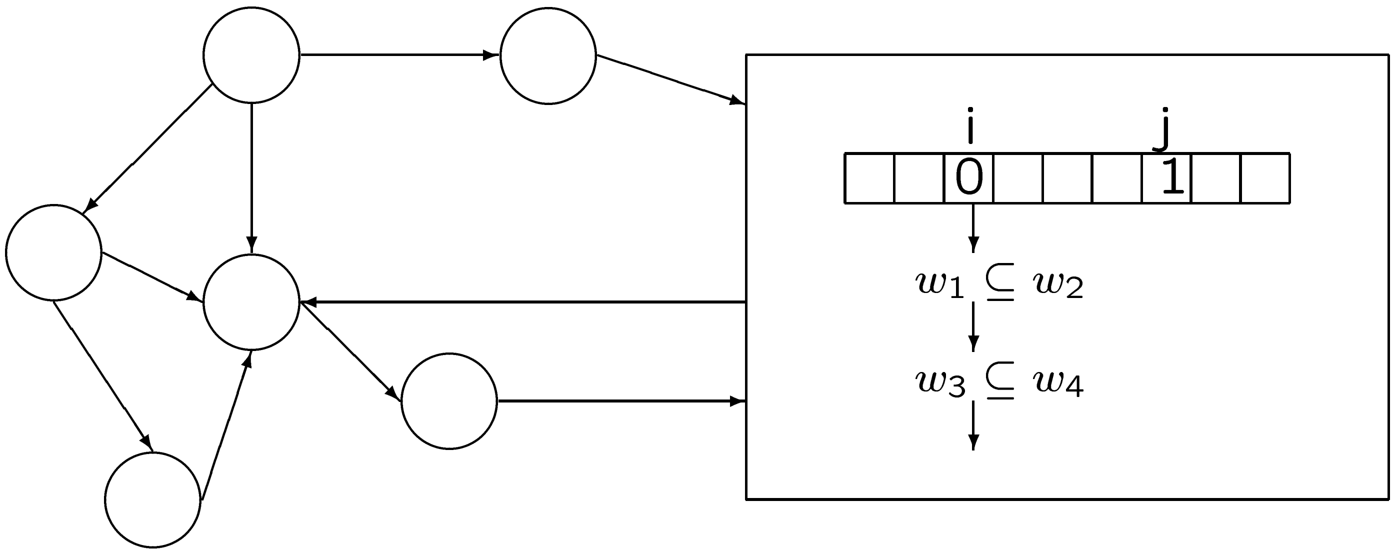


- Use clever data structures.

The 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 $v \rightarrow w$ 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



- $B(X)$ = *bitvector in node* X

- $K(X, i)$ = *list of pending constraints for bit* i *in node* X

Solver Operations

```
insert(C ⊆ X) =
   for i ∈ C do propagate(X,i) end


insert(X ⊆ Y) =
   create an edge X → Y;
   for i ∈ B(X) do propagate(Y,i) end


insert(c ∈ X ⇒ Y ⊆ Z) =
   if B(X,c) then
    insert(Y ⊆ Z)
   else
    K(X,c) := K(X,c)++(Y ⊆ Z)
   end


propagate(v,i) =
   if ¬B(v,i) then
    B(v,i) := true;
    for v → w do propagate(w,i) end;
    for k ∈ K(v,i) do 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.

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.

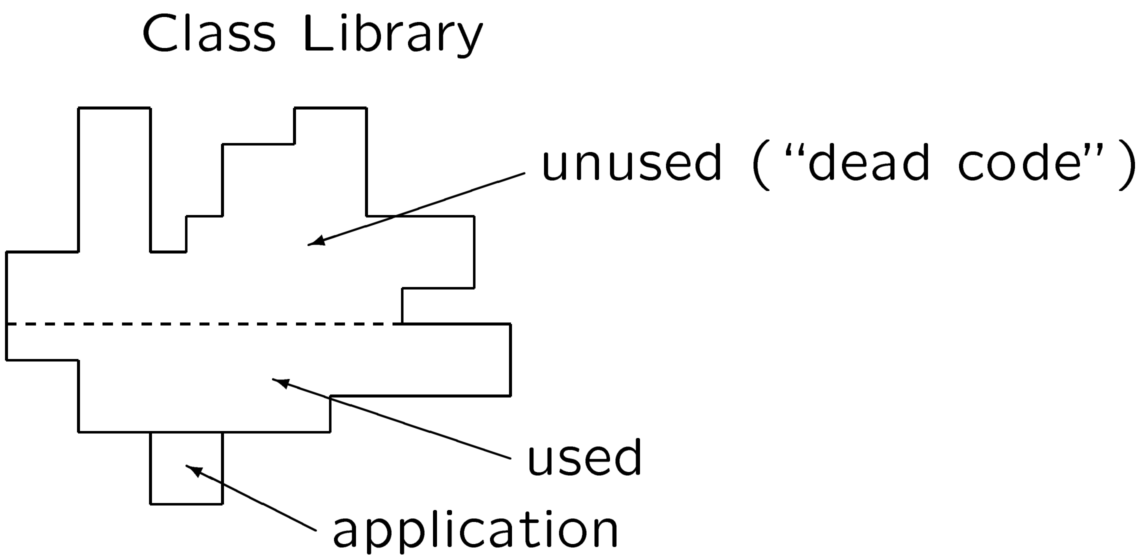
- Others require insight into the specific language in question.

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 := $\emptyset$;
  for $k \in$ constraints(main) **do**
   insert(k)
  end;
  while L$\neq$live **do**
   choose m from live$\backslash$L;
   for $k \in$ constraints(m) **do**
    insert(k);
    L := L$\cup$\{m\}
   end
  end;
  remove all methods not in L

- It is still $O(n^3)$.

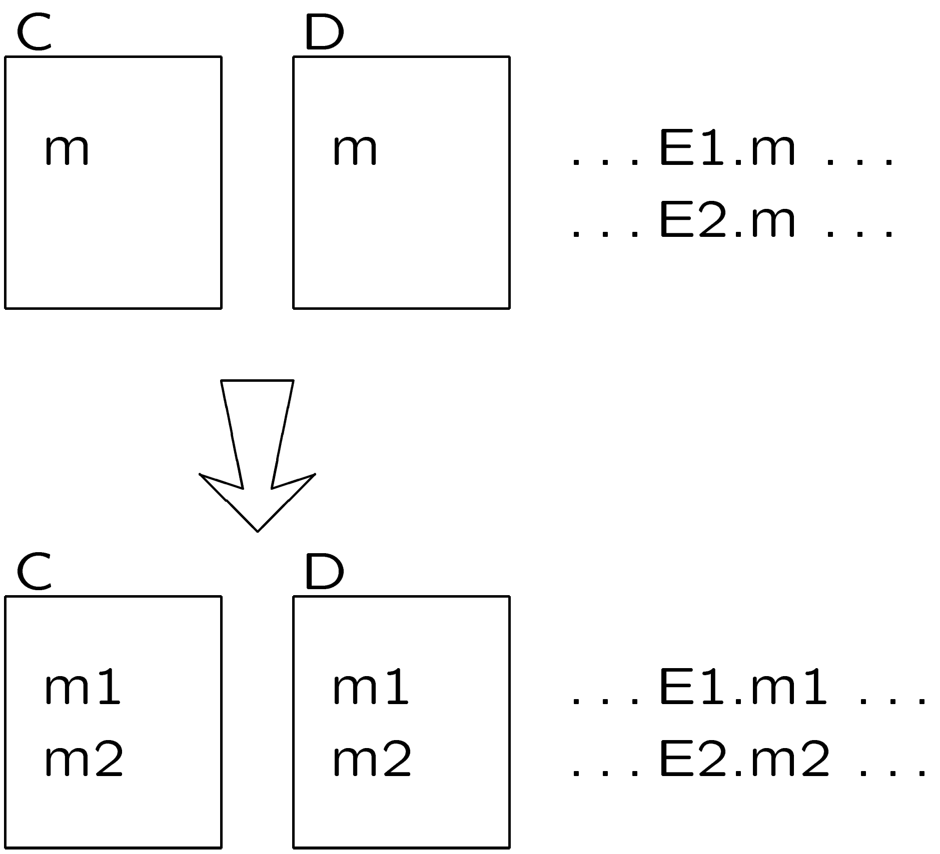
- The types of live methods may be improved.

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

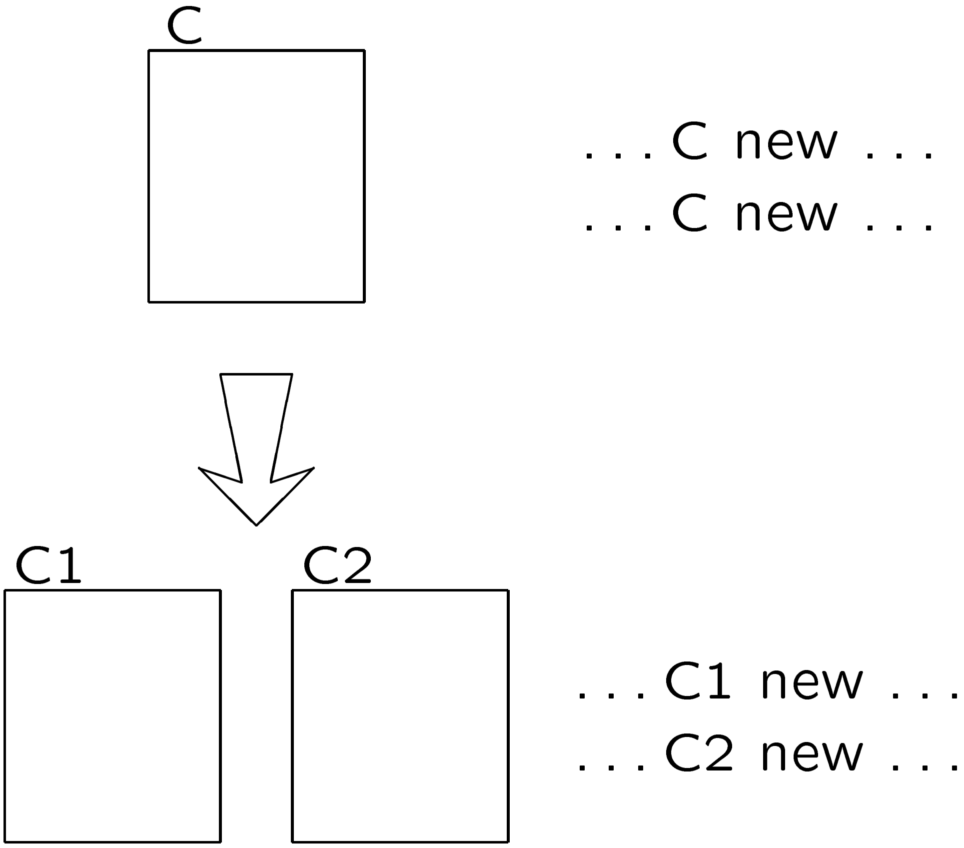
((C new).id@1 (7))+10;
((C new).id@2 (true)) or false
```

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.



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.

Implementation

- There are several full-scale implementations:
  - SELF (Stanford)
  - Concurrent Aggregates (Illinois)
  - BETA (Aarhus)
  - SMALLTALK? (OTI/Victoria)

- Each team contributes to the general technique.

- And solves language specific problems.

- Each language gains different advantages.

- General slogan: type inference helps you to avoid paying the price for general language constructs.

SELF

- Ole Agesen (Stanford/Sun Labs).
- Language features:
  - Prototypes, cloning and delegation.
  - Dynamic and multiple inheritance.
  - Blocks.
  - Dynamically typed.
- First full-scale implementation.

General Contributions

- The notion of *templates* to describe and compare various copying strategies.
- The Cartesian Product Algorithm: an elegant and powerful strategy for polymorphic methods.
- Termination of adaptive strategies.
- Techniques for blocks and closures.
- Preliminary results on incrementality.
- The implementation is integrated into an experimental system.
- Demonstrates proof-of-concept for a number of derived tools.
- Analyzes 5000 line `diff` tool in 38 seconds.

Applications

- Safety and ambiguity checker.
- Browser.
- Extractor (`diff`: 10M $\rightarrow$ 800K).
- Compiler.

Concurrent Aggregates

- John Plevyak and Andrew Chien (Illinois).
- Language features:
  - Concurrent and distributed objects.
  - First-class continuations.
  - Inheritance and delegation.
  - Dynamically typed.
- First implementation by "outsiders".

General Contributions

- The Iterative Algorithm: a useful strategy for both polymorphic methods and classes.
- Techniques for continuations.
- Techniques for concurrency.
- The implementation is integrated into a standard release.
- Basis for essential optimizations.
- Analyzes 2000 line network simulator in 200 seconds.

Applications

- Sequential optimizations:
  - Inlining and customization.
  - A host of traditional compiler optimization based on call graph approximated by type inference.

- Concurrent optimizations:
  - Removing unnecessary locks.
  - Hybrid execution model.
  - Locality optimization.

BETA

- Jørgen Greve and Jens Olsen (Aarhus).
- Language features:
  - Patterns unifying classes, methods, and blocks.
  - Generalized notion of virtuality.
  - Nested patterns.
  - Statically and dynamically typed.
- First implementation for a typed language.

General Contributions

- Integration of type inference and static type system.

- Adaptation of Cartesian Product Algorithm for nesting.

- Elegant and efficient technique for 1-level polymorphic classes.

- The implementation exists only as a preprocessor.

- Non-integrated tools are built.

- Analyzes 7000 line news index generator in 74 seconds.

Applications

- Extraction (news index generator: $\rightarrow$50%).

- Elimination of `QUA` checks (typically $>$90%).

- Inlining and customization (static dispatch $>$50%).

- Pattern classification:

  - *Abstract:* never instantiated, but subpattern is.

  - *Function*: don't refer to enclosing patterns.

  - *Class*: never evaluated.

  - *Method*: never referred by variables.

- Methods may be stack allocated.

SMALLTALK

- Bjorn Freeman-Benson, OTI/Victoria.

- Investigating type inference for full SMALLTALK.

- First industrial implementation.

- The usual goals:

  - optimizing compiler,

  - application packaging, and

  - browsing.

- No results as yet.

Modularity?

- The Holy Grail of global program analysis.

- Still an open problem.

- In practice: simplify and save generated constraints for a module in compact form along with partial solution.

- Additional code can only increase the types, so iterate further.

- Try to trace back through the constraints to limit the affected part.

- So far, only little experience.

Summary

- Type inference by:
  - *solving* constraints.
  - fast iterative algorithm.
- Improvements by:
  - trace graph.
  - code copying.
- Has been picked up by several real-life languages.
- Lifts the performance penalties of having general language constructs.
- Useful even if you like your own type system better.

Interfaces

Interfaces

- An interface tells *What*, not How.

- It lists *Services*, not Implementation.

- For BOPL, an interface lists method headers and, recursively, the interfaces for arguments and the return value.

- Grammar for interfaces:

Interface	::=	**Int**	
	|	**Bool**	

	|	**interface**
		   method Identifier (Interface,. . .)
		      returns Interface
		   . . .
		end	

	|	**Rec**(α) Interface	– recursive interface
	|	α	– interface variable

Examples

- **nil** has the interface **interface end**.

- 5 has the interface **Int**.

- Instances of the class

  class C
      var x
      method m(a)
          not a; 5
      end
  end

  have the interface

  interface
      method m(**Bool**) **returns Int**
  end

Recursive Interfaces

- Consider the annotated class

```
class C
   method m() returns {C}
      self
   end
end
```

- It has the interface

```
interface
   method m() returns
      interface
         method m() returns
            ...
end
```

- An infinite interface!

- We will write it as:

```
Rec(α) interface
   method m() returns α
end
```

Subtyping for Interfaces

- [Amadio and Cardelli, ACM TOPLAS 1993]

- Intuitively, a big interface is a subtype of a small interface.

- The interface

  > **interface**
  >   **method** m(**Bool**) **returns Int**
  >   **method** p(**Int**) **returns Int**
  > **end**

  is a subtype of

  > **interface**
  >   **method** m(**Bool**) **returns Int**
  > **end**

Covariance and Contravariance

- Contravariance for arguments; covariance for results.

- The interface

  interface
      method m(**S**) **returns T**
      method p(**Int**) **returns Int**
  end

  is a subtype of

  interface
      method m(**S'**) **returns T'**
  end

  if **S'** is a subtype of **S**,
  and **T** is a subtype of **T'**.

Deciding Subtyping

- Care must be taken with recursive interfaces.

- For recursive interfaces, subtyping can be decided in $O(n^2)$ time.
  [Kozen, Palsberg, Schwartzbach,
  ACM POPL 1993, MSCS 1995]

From Sets of Classes to Interfaces

- We have an algorithm for inferring sets of classes.

- We will now give a "backend" which does

  Sets of Classes $\rightarrow$ Interfaces

- Based on
  [Palsberg and O'Keefe, ACM TOPLAS 1995].

- Worst case: exponential time.

- Net result: an interface-inference algorithm!

Simple Example

From the program

```
class C
   var x
   method m(a)
      not a; 5
   end
end
```

we get the constraints

$$
\begin{aligned}
[\![\mathsf{a}]\!] &= \{\mathbf{Bool}\} \\
[\![\textbf{not}\ \mathsf{a}]\!] &= \{\mathbf{Bool}\} \\
[\![\mathsf{5}]\!] &= \{\mathbf{Int}\} \\
[\![\textbf{not}\ \mathsf{a};\ \mathsf{5}]\!] &= [\![\mathsf{5}]\!]
\end{aligned}
$$

which have the solution φ where

$$
\begin{aligned}
\varphi([\![\mathsf{a}]\!]) &= \varphi([\![\textbf{not}\ \mathsf{a}]\!]) = \{\mathbf{Bool}\} \\
\varphi([\![\mathsf{5}]\!]) &= \varphi([\![\textbf{not}\ \mathsf{a};\mathsf{5}]\!]) = \{\mathbf{Int}\}
\end{aligned}
$$

So, instances of C have the interface

```
interface
   method m(Bool) returns Int
end
```

Interface Constructors

- The class

```
class C
   var x
   method m(a)
      not a; 5
   end
end
```

yields the interface constructor

```
interface
   method m(•) returns •
end
```

- Building the interface as a tree:

$$\begin{array}{c}\textbf{interface method } \mathrm{m}(\underset{\downarrow}{\bullet}) \textbf{ returns } \underset{\downarrow}{\bullet} \textbf{ end}\\ \qquad\qquad\qquad\qquad \textbf{Bool} \qquad\quad \textbf{Int}\end{array}$$

Representing Interfaces

- Issue: recursive interfaces.
- Solution: backpointers.
- Result: the representation is an automaton.
- The start state is the “root” of the interface.
- The words accepted by the automaton are “paths” in the interface.
- The labels of states in the automaton are interface constructors.
- The automaton:

  m.returns **interface method** m() **returns** • **end**

  represents the recursive interface from before.

The Automaton

- Given "sets of classes", we will in the general case construct an automaton (representing an interface) where every state is a set of classes.

- The automaton is exponentially big!

- Reason: There are exponentially many subsets of the set of classes in the program.

- For every set of classes, we label the corresponding state with the "biggest common interface constructor" (or with "error" if there is a conflict).

The Automaton

- Let φ be a solution to the type inference problem. So, φ maps each $[\![\ldots]\!]$ to a set of classes.

- If we want to know an interface for the expression E, then let the start state of the automaton be $\varphi([\![E]\!])$.

- All states accept.

- Transition function δ. Suppose the biggest common interface constructor for $\{C_1, \ldots, C_n\}$ has a method m.

$$\delta(\{C_1, \ldots, C_n\}, \text{m.return}) = \bigcup_{i=1}^{n} \varphi([\![E_i]\!])$$

  where E_i is the body of method m in class C_i.

- Similar definition for the arguments, but with $\cap$ instead of $\cup$.

Summary

- We can infer interfaces in two steps:

    1. Infer sets of classes.

    2. Construct interfaces from those sets.

- Theorems and examples (for λ-calculus) are in [Palsberg & O'Keefe, ACM TOPLAS 1995].

Literature

Literature

Related literature by Palsberg and Schwartzbach:

- **Object-Oriented Type Systems**, Wiley & Sons, 1994, ISBN 0 471 94128 X.
- **Static Typing for Object-Oriented Programming**, in *Science of Computer Programming, 23(1)*, 1994.
- **Type Substitution for Object-Oriented Programming**, in *OOPSLA/ECOOP'90*, ACM, 1990.
- **What is Type-Safe Code Reuse?**, in *Proc. ECOOP'91*, Springer-Verlag, 1991.
- **Object-Oriented Type Inference**, in *Proc. OOPSLA'91*, ACM, 1991.
- **Three Discussions on Object-Oriented Typing**, *OOPS Messenger, 3(2)*, ACM, 1992.
- **Making Type Inference Practical**, with Nicholas Oxhøj, in *Proc. ECOOP'92*, Springer-Verlag, 1992.
- **Type Inference of SELF**, with Ole Agesen, in *Software – Practice & Experience, 25(9)*, 1995. Preliminary version in Proc. ECOOP'93.

An annotated bibliography on OO type systems is found at:

`http://www.daimi.aau.dk/~mis/popl94.ps`

Literature

Other literature on type inference:

- **The Cartesian Product Algorithm**, Ole Agesen, in *Proc. ECOOP'95*, Springer-Verlag, 1995.
- **Type Inference of BETA**, Jørgen Greve, Jens Cornelius Olsen, *Master's Thesis*, University of Aarhus, 1995.
- **Constraint-Based Type Inference and Parametric Polymorphism**, Ole Agesen, in *SAS'94, LNCS 864*, Springer-Verlag, 1994.
- **Sifting Out the Gold**, Ole Agesen and Dave Ungar, in *Proc. OOPSLA'94*, ACM, 1994.
- **Precise Concrete Type Inference for Object-Oriented Languages**, John Plevyak and Andrew Chien, in *Proc. OOPSLA'94*, ACM, 1994.
- **Compile-Time Analysis of Object-Oriented Programs**, Jan Vitek, Nigel Horspool, J.S. Uhl, in *CC'92, LNCS 641*, Springer-Verlag, 1992.
- **A Type System Equivalent to Flow Analysis**, Jens Palsberg and Patrick O'Keefe, in *ACM TOPLAS, 17(4)*, 1995.
- **Simple and Effective Analysis of Statically Typed Object-Oriented Programs**, Amer Diwan, J. Eliot B. Moss, K. S. McKinley, in OOPSLA'96, ACM, 1996.
- And many more!