Modular Dataflow Analysis

Aivar Annamaa
Feb. 23rd, 2010

Based on:
Rountev, Sharp, Xu, 2008
„IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries“

Problem

- Interprocedural analyses are usually too slow
  - can take many hours
  - can take many seconds (not usable „as-you-type“)

- If it's fast enough then probably not very precise

Solutions?

- Reduce precision?
  - can make analysis useless/unusable

- Go modular
  - analyze each part (eg. method) independently
  - analysis process could be parallelized
  - cache results (method summaries)
  - only changed methods need to be re-analyzed

Challenges for modularity

- Dependencies between parts
- How to represent method summaries?

Agenda

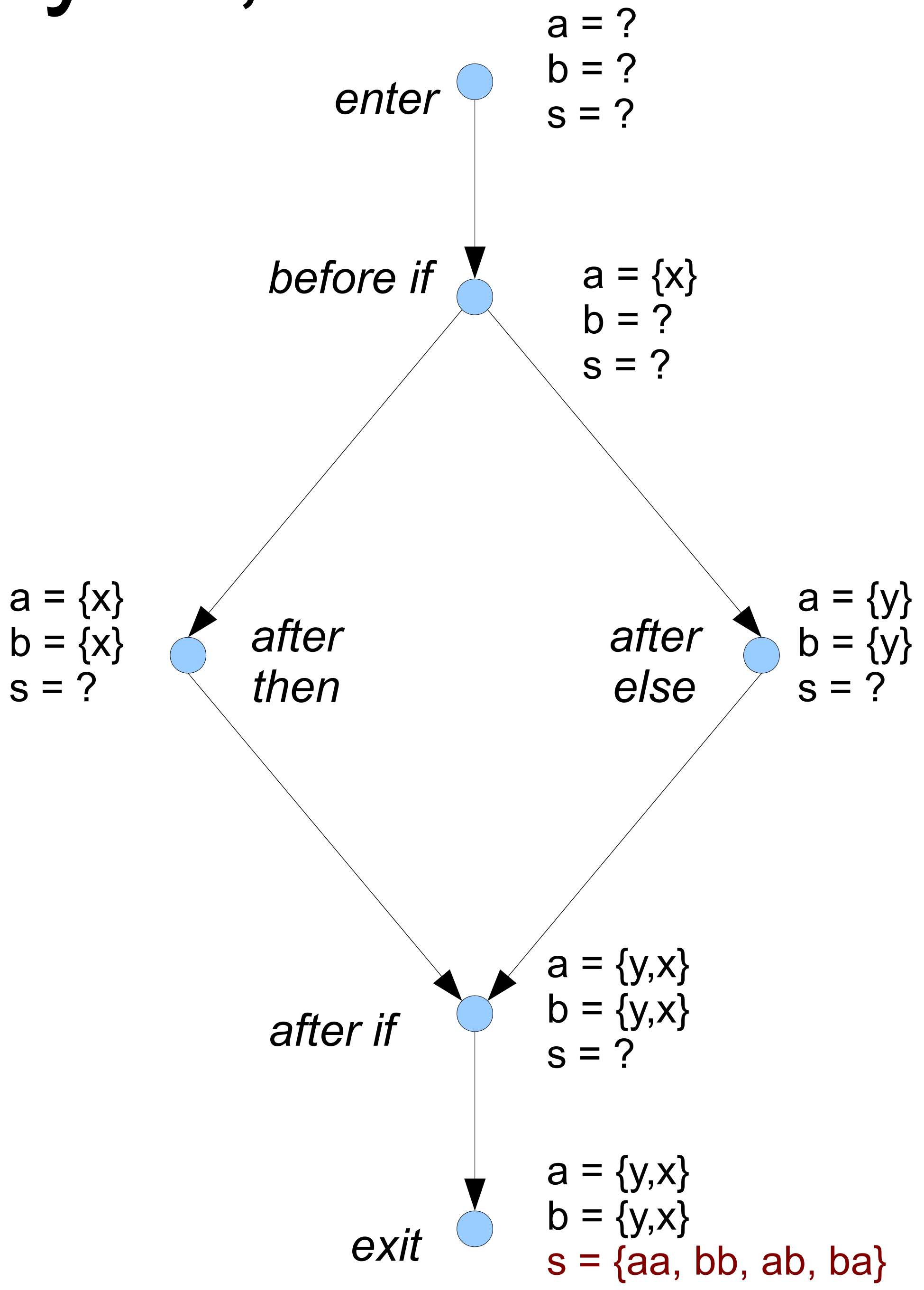
- Dataflow analysis
- An approach for solving IDE problems
    - IDE
    - Transformers as graphs
    - Example analysis
    - Summary generation
    - Benchmarks and conclusions

Dataflow analysis, CFG

```
a = „x“

if aCondition()
{
     b = „x“
}
else {
     a = „y“
     b = „y“
}

s = a + b
```



Lattice of abstract values

- Elements are partially ordered
- *x* ≤ *y* means *y* is as least as precise as *x*
- two values are combined with meet (or *glb*) operator ∧
- on picture ∧ = ∪ and ≤ = ⊇
- can be used for env-s

$$\begin{tikzcd}
 & \{\} \arrow[dl] \arrow[d] \arrow[dr] & & (\top) \\
\{d_1\} \arrow[d] \arrow[dr] & \{d_2\} \arrow[dl] \arrow[dr] & \{d_3\} \arrow[dl] \arrow[d] & \\
\{d_1, d_2\} \arrow[dr] & \{d_1, d_3\} \arrow[d] & \{d_2, d_3\} \arrow[dl] & \\
 & \{d_1, d_2, d_3\} & & (\bot)
\end{tikzcd}$$

CFG, environments, transformers

- Each CGF node has environment representing *dataflow facts*
  - $env :: D \rightarrow L$
  - D = set of variables
  - L = set of abstract values
- Each edge has transformer
  - $t :: env \rightarrow env$
- CFG + variables + lattice + transformers = abstract version of the program

Solving dataflow problem

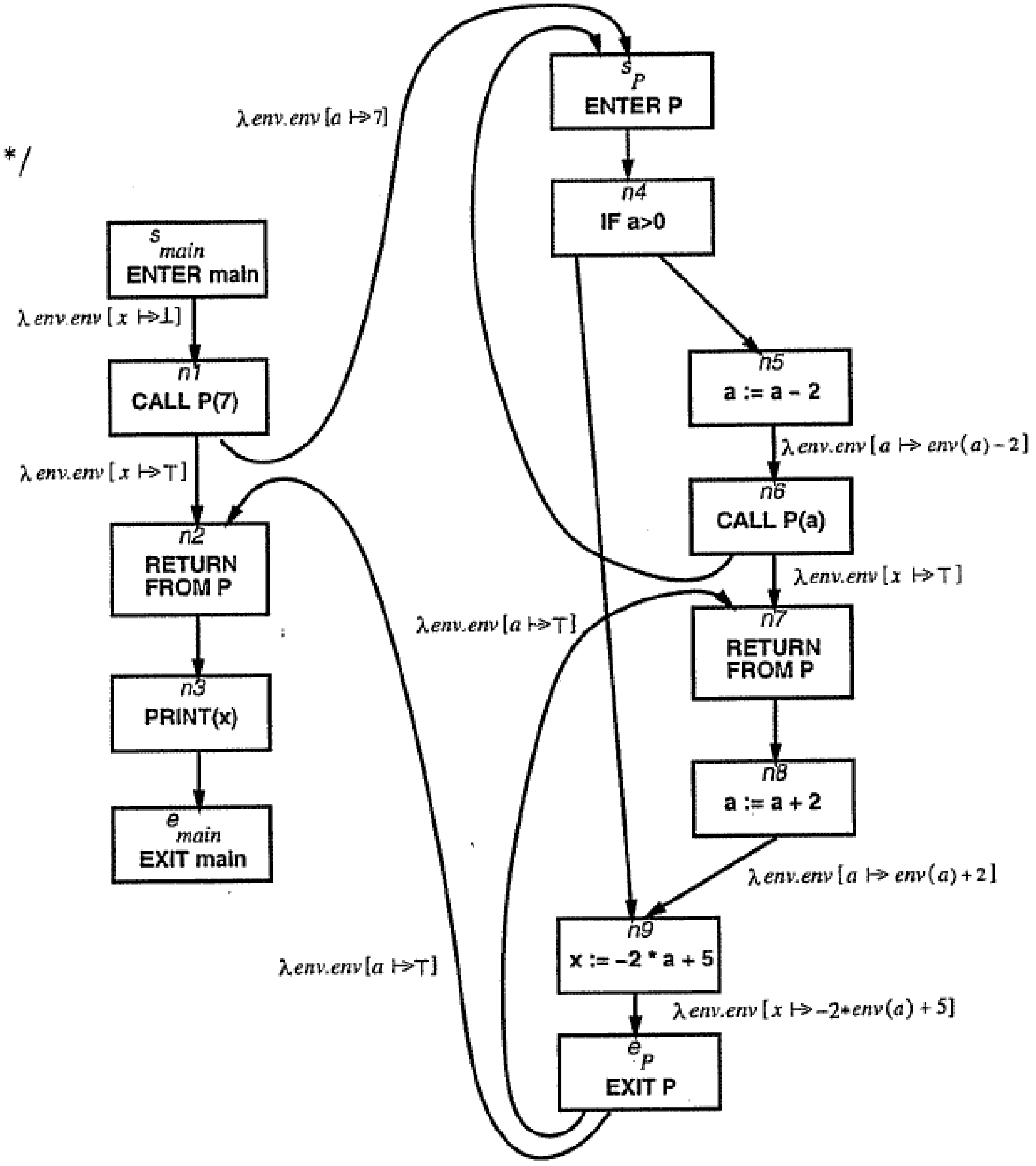
- Forward analysis
  - start from entry node and propagate values downward
- Backward analysis
  - start from exit and move upwards
- Cycles in CFG complicate things
  - loop until transformers don't change anything
  - often requires certain tricks to ensure termination

Interprocedural dataflow analysis

- How to handle method calls?
- Inlining called methods
  - Good: it's precise
  - Bad: graph can grow huge
  - Bad: doesn't work with recursion
- Extend CFG
  - add call nodes
  - add return nodes

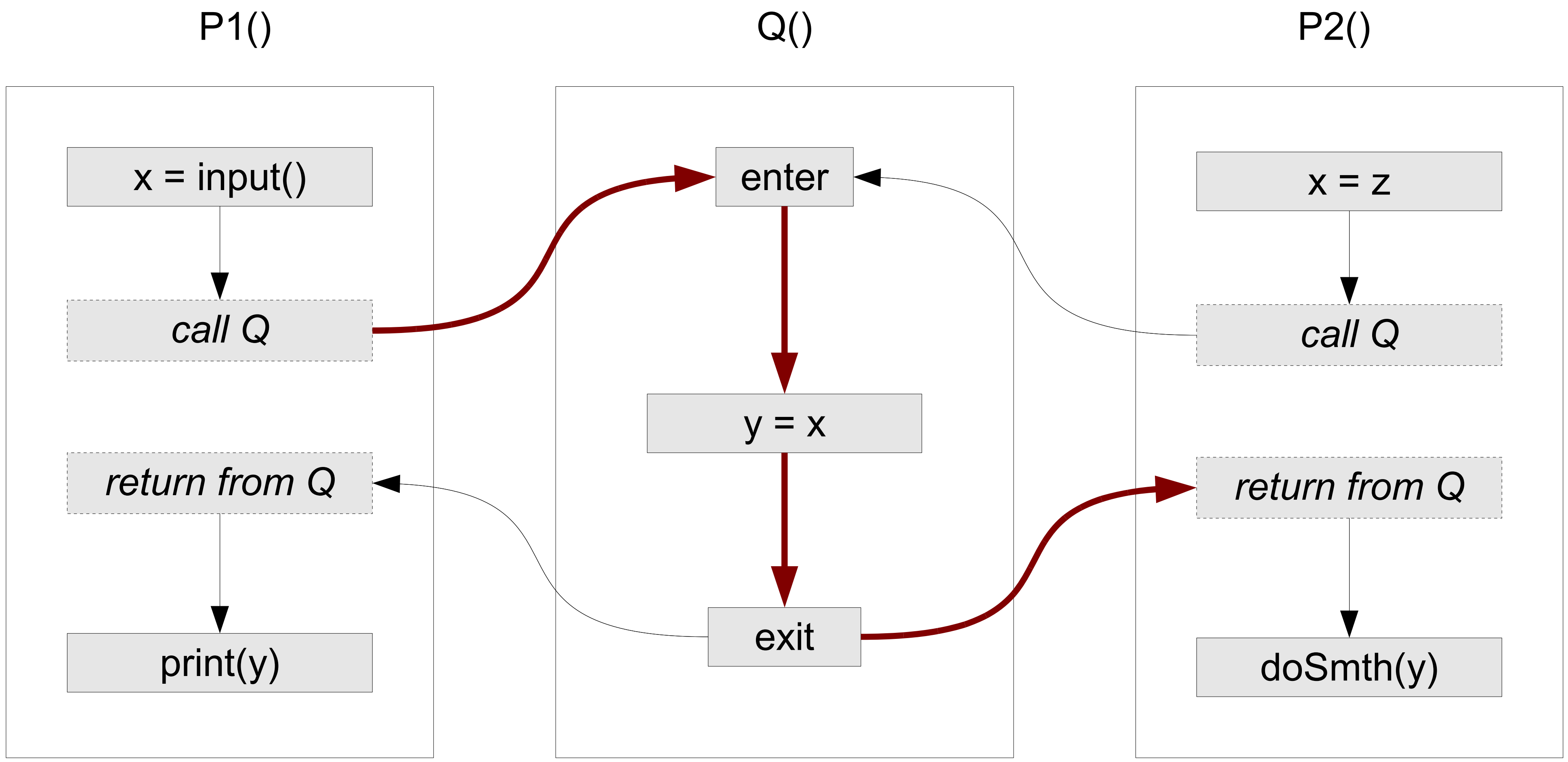
```
declare x: integer
program main
begin
        call P(7)
        print (x) /* x is a constant here */
end

procedure P (value a : integer)
begin /* a is not a constant here */
        if a > 0 then
           a := a − 2
           call P (a)
           a := a + 2
        fi
        x := −2 * a + 5
        /* x is not a constant here */
end
```



s_P ENTER P
$n4$ IF a>0
s_{main} ENTER main
$\lambda env.env[x \mapsto \perp]$
$n1$ CALL P(7)
$\lambda env.env[a \mapsto 7]$
$\lambda env.env[x \mapsto \top]$
$n2$ RETURN FROM P
$n3$ PRINT(x)
e_{main} EXIT main
$n5$ a := a - 2
$\lambda env.env[a \mapsto env(a)-2]$
$n6$ CALL P(a)
$\lambda env.env[x \mapsto \top]$
$\lambda env.env[a \mapsto \top]$
$n7$ RETURN FROM P
$n8$ a := a + 2
$\lambda env.env[a \mapsto env(a)+2]$
$\lambda env.env[a \mapsto \top]$
$n9$ x := -2 * a + 5
$\lambda env.env[x \mapsto -2*env(a)+5]$
e_P EXIT P

Unrealizable paths



Conclusion of introduction

- D = variables
- L = abstract values (in form of lattice)
- env :: $D \to L$ = dataflow facts
  - Env($D \to L$) = lattice of all such environments
- CFG as abstract program
  - Dataflow facts in nodes
  - Environment transformers on edges
- Interprocedural = trouble

IDE Dataflow Problems

- Interprocedural Distributive Environment
- program is represented by ICFG
- dataflow facts are environments $D \rightarrow L$ mapping variables to some abstract values
- L is semi-lattice of finite height
- transformers are distributive
  - $t\,(env_1 \wedge env_2) = t\,(env_1) \wedge t\,(env_2)$

Example: Dependence analysis

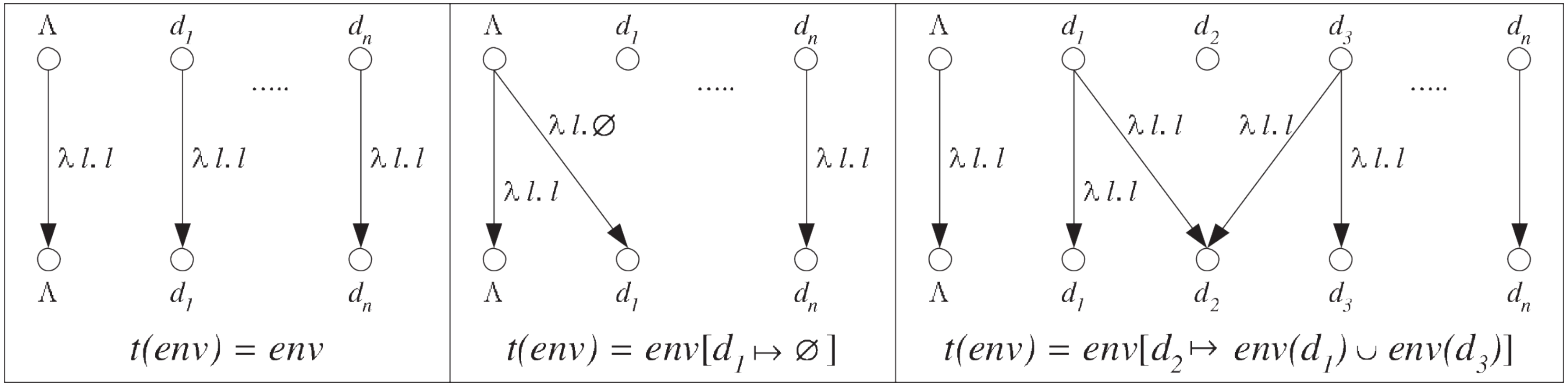
- Which parameters influence a variable?
- Flow-sensitive
- D = all local variables and formal parameters
- L = powerset of formal parameters
  - with partial order $\supseteq$ and meet $\cup$

Dependece analysis. Transformers

- $d_2 = d_1 + d_3$;
  - $env[d_1 \rightarrow env(d_1)$ □ $env(d_3)]$
- $d_1 = 68$
  - $env[d_1 \rightarrow \emptyset]$
- $d = f(d_1, d_2)$
  - assign actual arguments to formal parameters
  - use *f*'s *summary function*
  - assign result value to *d*

Transformers as graphs

print(68) $d_1 = 68$ $d_2 = d_1 + d_3$



Λ
d_1
d_2
d_3
d_n
.....
$\lambda l.l$
$\lambda l.\varnothing$
$t(env) = env$
$t(env) = env[d_1 \mapsto \varnothing]$
$t(env) = env[d_2 \mapsto env(d_1) \cup env(d_3)]$

- transformer functions are given *pointwise*
- Λ represents „something else than a variable“
- meet = graph union
  composition = graph transitive closure

Type analysis

- „0-CFA type analysis“
- What type can a variable possibly be?
- Relevant in OO because of polymorphism
- D = vars, params (incl. this), fields
- L = powerset of all types

Type Analysis 2

- d := new T
  - env [d $\rightarrow$ env(d) $\cup$\{T\}]
- d_1 := d_2
  - env [d_1 $\rightarrow$ env(d_1) $\cup$env(d_2)]
  - Flow insensitive
    - each transform can make result only less precise
- d_1 = d_2.m()
  - env [d_1 $\rightarrow$ [t (x.m()) | x $\in$ env(d_2)]]

Different calls and methods

- Exit calls
  - method is not statically known
  - „exits“ the scope of analysis and can't be modeled in advance
- Fixed calls
  - only one possible target method
  - eg. static methods on final classes
- Fixed methods
  - has only fixed calls in it

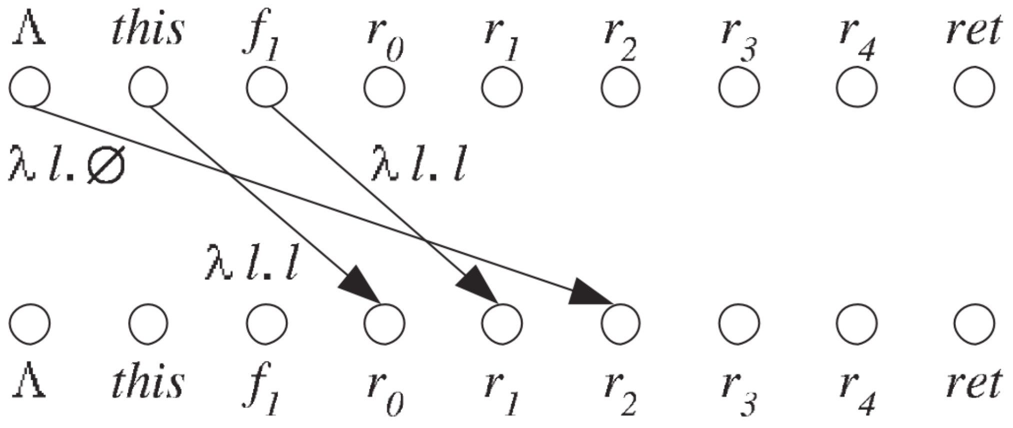
Method summary generation

- Summary uses graph representation
- At method calls:
  - fixed calls to fixed methods
    - inline method summary
  - other calls
    - insert placeholder
    - resolved at full program analysis
- Summary is abstracted
  - irrelevant details (for summary clients) are removed

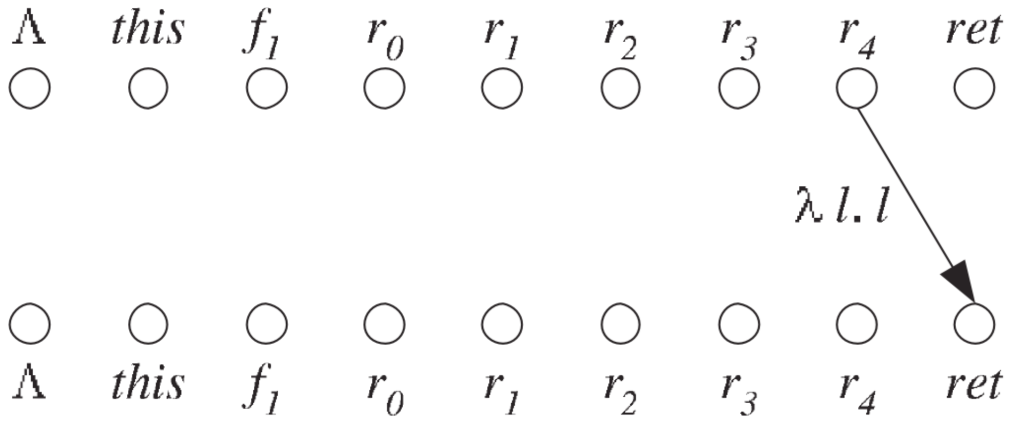
Example of Dependency Analysis

```
class DateFormat
String format(Date f1) {
       DateFormat r0; Date r1;
       StringBuffer r2, r3;
       r0 = this; r1 = f1;
       r2 = new StringBuffer();
cs1:  r3 = r0.format(r1,r2);
cs2:  String r4 = r3.toString();
       return r4; }
abstract StringBuffer format
(Date,StringBuffer);

subclass SimpleDateFormat
StringBuffer format
(Date f2,StringBuffer f3) {...}
```



(a) Transformer for entry → cs1

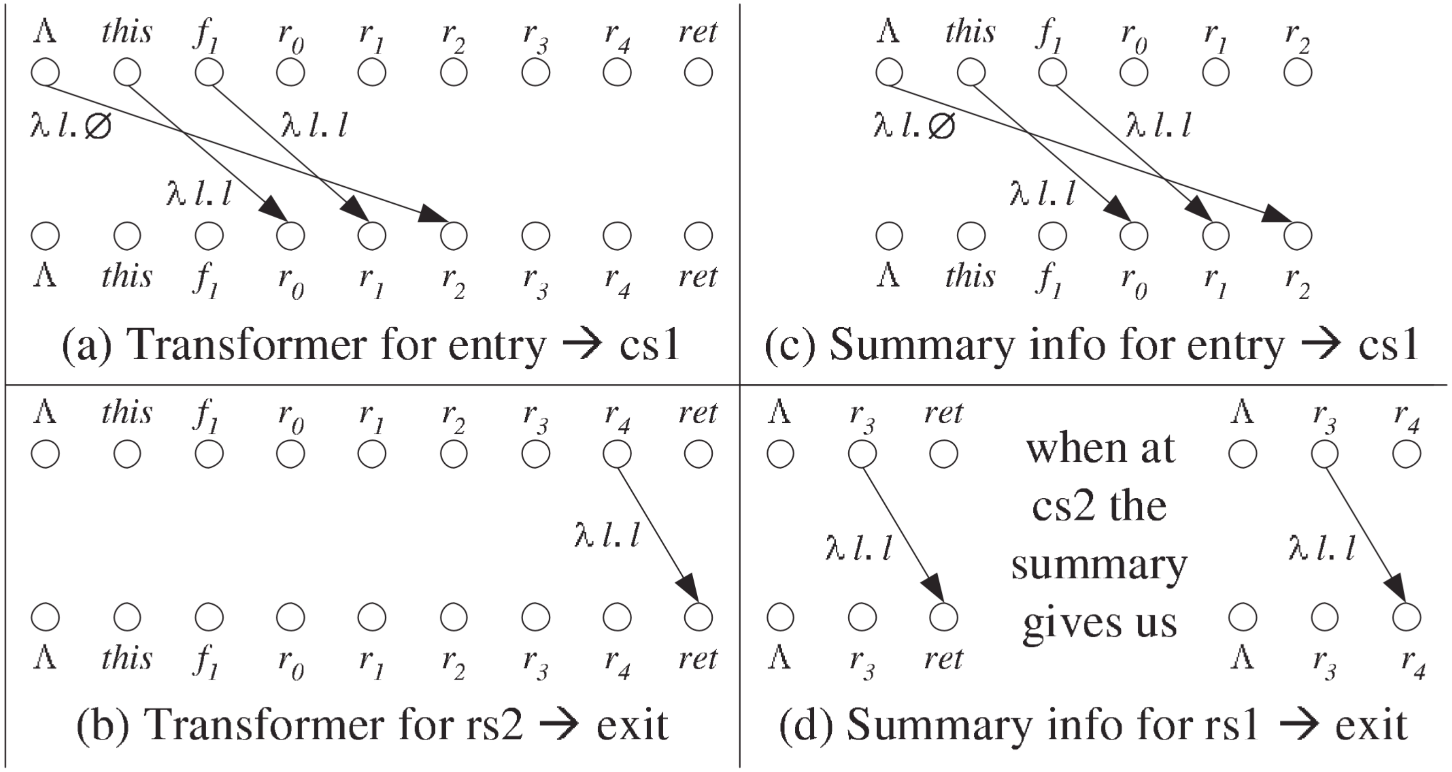


(b) Transformer for rs2 → exit

Example summary graph

```
class DateFormat
String format(Date f1) {
        DateFormat r0; Date r1;
        StringBuffer r2, r3;
        r0 = this; r1 = f1;
        r2 = new StringBuffer();
cs1:  r3 = r0.format(r1,r2);
cs2:  String r4 = r3.toString();
        return r4; }
abstract StringBuffer format
(Date,StringBuffer);

subclass SimpleDateFormat
StringBuffer format
(Date f2,StringBuffer f3) {…}
```


Λ this f_1 r_0 r_1 r_2 r_3 r_4 ret
λ l.∅
λ l. l
λ l. l
Λ this f_1 r_0 r_1 r_2 r_3 r_4 ret
Λ this f_1 r_0 r_1 r_2
λ l.∅
λ l. l
λ l. l
Λ this f_1 r_0 r_1 r_2
Λ this f_1 r_0 r_1 r_2 r_3 r_4 ret
λ l. l
Λ this f_1 r_0 r_1 r_2 r_3 r_4 ret
Λ r_3 ret
λ l. l
Λ r_3 ret
when at cs2 the summary gives us
Λ r_3 r_4
λ l. l
Λ r_3 r_4

(a) Transformer for entry → cs1

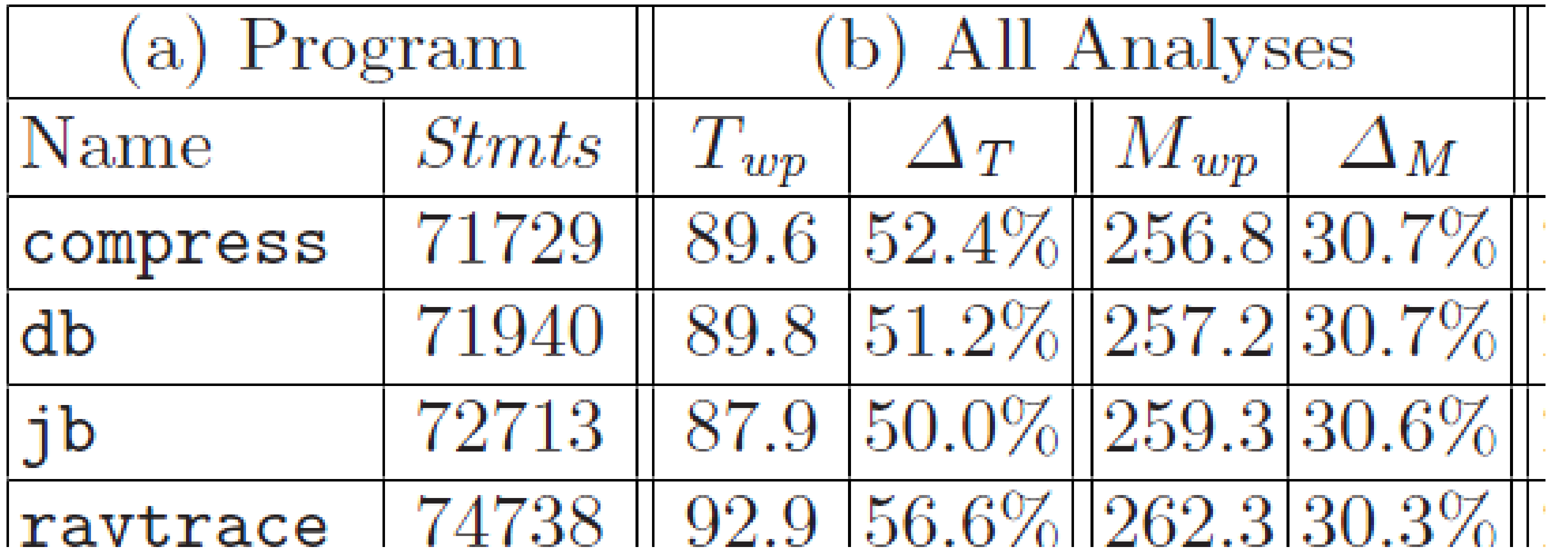
(c) Summary info for entry → cs1

(b) Transformer for rs2 → exit

(d) Summary info for rs1 → exit

Experimental evaluation

- Created summaries for Java 1.4 (25490 methods)
- 33% of the methods are fixed
- Summaries used for analyzing 20 programs



(a) Program		(b) All Analyses			
Name	*Stmts*	T_{wp}	Δ_T	M_{wp}	Δ_M
`compress`	71729	89.6	52.4%	256.8	30.7%
`db`	71940	89.8	51.2%	257.2	30.7%
`jb`	72713	87.9	50.0%	259.3	30.6%
`raytrace`	74738	92.9	56.6%	262.3	30.3%

Conclusion

- Transfer functions can be efficiently represented as graphs
- Summaries of these method graphs can be reused on different call sites
  - Fixed calls are common enough to deserve special optimisations (inlining)
- Analyses with precomputed library summaries are 2x faster than analyses „from scratch“

References

- Rountev, Sharp, Xu, 2008
  „IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries“
- Sagiv, Reps, Horwitz, 1996
  „Precise interprocedural dataflow analysis with applications to constant propagation“
- Cousot & Cousot, 2002
  „Modular Static Program Analysis“