Optimizing Compilers

Inter-Procedural Dataflow Analysis

Markus Schordan

Institut für Computersprachen
Technische Universität Wien

Syntax

$$P_\star \;::=\; \text{begin } D_\star\ S_\star \text{ end}$$
$$D \;::=\; D; D \mid \text{proc } p(\text{val } x;\ \text{res } y) \text{ is}^{\ell_n}\ S \text{ end}^{\ell_x}$$
$$S \;::=\; \dots \mid [\text{call } p(a, z)]^{\ell_c}_{\ell_r}$$

Labeling scheme

- procedure declarations
  - ℓ_n: for entering the body
  - ℓ_x: for exiting the body
- procedure calls
  - ℓ_c: for the call
  - ℓ_r: for the return

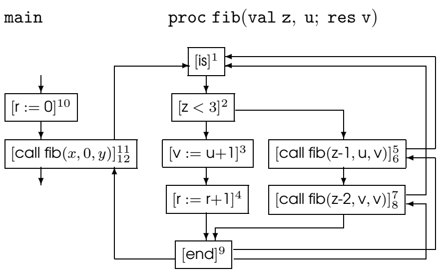
Analysing Procedures

We consider procedures with call-by-value and call-by-result parameters.

Example:

```
begin
  proc fib(val z,u; res v) is
    if z<3 then
      (v:=u+1; r:=r+1)
    else (
      call fib (z-1,u,v);
      call fib (z-2,v,v)
      )
  end;
r:=0;
call fib(x,0,y)
end
```


Example Flow Graph



Flow Graph for Procedures

	$[\text{call } p(a,z)]^{\ell_c}_{\ell_r}$	$\text{proc } p(\text{val } x;\ \text{res } y) \text{ is}^{\ell_n}\ S \text{ end}^{\ell_x}$
init	ℓ_c	ℓ_n
final	$\{\ell_r\}$	$\{\ell_x\}$
blocks	$\{[\text{call } p(a,z)]^{\ell_c}_{\ell_r}\}$	$\{\text{is}^{\ell_n}\} \cup \text{blocks}(S) \cup \{\text{end}^{\ell_x}\}$
labels	$\{\ell_c, \ell_r\}$	$\{\ell_c, \ell_r\} \cup \text{labels}(S)$
flow	$\{(\ell_c; \ell_n), (\ell_x; \ell_r)\}$	$\{(\ell_n, \text{init}(S))\} \cup \text{flow}(S) \cup \{(\ell, \ell_x) \mid \ell \in \text{final}(S))\}$

- $(\ell_c; \ell_n)$ is the flow corresponding to calling a procedure at ℓ_c and entering the procedure body at ℓ_n and
- $(\ell_x; \ell_r)$ is the flow corresponding to exiting a procedure body at ℓ_x and returning to the call at ℓ_r.

Naive Formulation

Treat the three kinds of flow, (ℓ_1, ℓ_2), $(\ell_c; \ell_n)$, $(\ell_x; \ell_r)$ in the same w

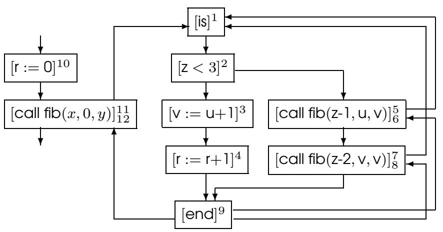
Equation system:

$$A_\circ(\ell) \;=\; \bigsqcup\{A_\bullet(\ell') \mid (\ell', \ell) \in F \vee (\ell'; \ell) \in F\} \sqcup \iota^\ell_E$$
$$A_\bullet(\ell) \;=\; f^A_\ell(A_\circ(\ell))$$

- both procedure calls $(\ell_c; \ell_n)$ and procedure returns $(\ell_x; \ell_r)$ c treated like "goto's".
- there is no mechanism for ensuring that information flowing $(\ell_c; \ell_n)$ flows back along $(\ell_x; \ell_r)$ to the *same* call
- intuitively, the equation system considers a much too large s "paths" through the program and hence will be grossly imp (although formally on the safe side)

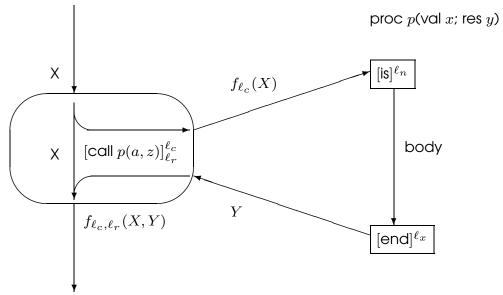
Matching Procedure Entries and Exits

`main` `proc fib(val z, u; res v)`



We want to overcome the shortcoming of the naive formulation by restricting attention to paths that have the proper nesting of procedure calls and exits.

General Formulation: Calls and Returns



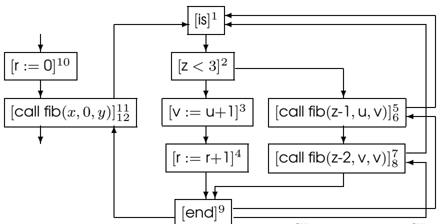
"Meet" over Valid Paths (MVP)

A complete path from ℓ_1 to ℓ_2 in $P_\star$ has proper nesting of proce entries and exits; and a procedure returns to the point where it called:

$CP_{\ell_1,\ell_2} \longrightarrow \ell_1$	whenever $\ell_1 = \ell_2$
$CP_{\ell_1,\ell_3} \longrightarrow \ell_1, CP_{\ell_2,\ell_3}$	whenever $(\ell_1,\ell_2) \in$ flow$_\star$
$CP_{\ell_c,\ell} \longrightarrow \ell_c, CP_{\ell_n,\ell_x}, CP_{\ell_r,\ell}$	whenever$P_\star$ contains [call $p(a,z)$
	andproc p(val x; res y) is$^{\ell_n}$ S end

Definition: $(\ell_c,\ell_n,\ell_r,\ell_x) \in$ interflow$_\star$ if $P_\star$ contains [call $p(a,z)]^{\ell_c}_{\ell_r}$ c as proc p(val x; res y) is$^{\ell_n}$ S end$^{\ell_x}$

Example



$CP_{10,12} \rightarrow 10, CP_{11,12}$

$CP_{11,12} \rightarrow 11, CP_{1,9}, CP_{12,12}$

$CP_{1,9} \rightarrow 1, CP_{2,9}$

$CP_{2,9} \rightarrow 2, CP_{3,9}$

$CP_{2,9} \rightarrow 2, CP_{5,9}$

$CP_{3,9} \rightarrow 3, CP_{4,9}$

$CP_{4,9} \rightarrow 4, CP_{9,9}$

$CP_{5,9} \rightarrow 5, CP_{1,9}, CP_{6,9}$

$CP_{6,9} \rightarrow 6, CP_{7,9}$

$CP_{7,9} \rightarrow 7, CP_{1,9}, CP_{8,9}$

$CP_{8,9} \rightarrow 8, CP_{9,9}$

$CP_{12,12} \rightarrow 12$

$CP_{9,9} \rightarrow 9$

Some valid paths: (10,11,1,2,3,4,9,12) and (10,11,1,2,5,1,2,3,4,9,6,7,1,2,3,4,9,8,9,12)

A non-valid path: (10,11,1,2,5,1,2,3,4,9,12)

Valid Paths

A valid path starts at the entry node init$_\star$ of $P_\star$, all the procedure exits match the procedure entries but some procedures might be entered but not yet exited:

$VP_\star \longrightarrow VP_{\text{init}_\star,\ell}$	whenever $\ell \in$ Lab$_\star$
$VP_{\ell_1,\ell_2} \longrightarrow \ell_1$	whenever $\ell_1 = \ell_2$
$VP_{\ell_1,\ell_3} \longrightarrow \ell_1, VP_{\ell_2,\ell_3}$	whenever $(\ell_1,\ell_2) \in$ flow$_\star$
$VP_{\ell_c,\ell} \longrightarrow \ell_c, CP_{\ell_n,\ell_x}, VP_{\ell_r,\ell}$	whenever$P_\star$ contains [call $p(a,z)]^{\ell_c}_{\ell_r}$
	andproc p(val x; res y) is$^{\ell_n}$ S end$^{\ell_x}$
$VP_{\ell_c,\ell} \longrightarrow \ell_c, VP_{\ell_n,\ell}$	whenever$P_\star$ contains [call $p(a,z)]^{\ell_c}_{\ell_r}$
	andproc p(val x; res y) is$^{\ell_n}$ S end$^{\ell_x}$

MVP Solution

$$MVP_\circ(\ell) = \bigsqcup\{f_{\vec{\ell}}(\iota) | \vec{\ell} \in vpath_\circ(\ell)\}$$

$$MVP_\bullet(\ell) = \bigsqcup\{f_{\vec{\ell}}(\iota) | \vec{\ell} \in vpath_\bullet(\ell)\}$$

where

$vpath_\circ(\ell) = \{[\ell_1,\ldots,\ell_{n-1}] \mid n \geq 1 \wedge \ell_n = \ell \wedge [\ell_1,\ldots,\ell_n]$ is valid path$\}$

$vpath_\bullet(\ell) = \{[\ell_1,\ldots,\ell_n] \mid n \geq 1 \wedge \ell_n = \ell \wedge [\ell_1,\ldots,\ell_n]$ is valid path$\}$

The MVP solution may be undecidable for lattices satisfying the A ing Chain Condition, just as was the case for the MOP solution.

Making Context Explicit

- The MVP solution may be undecidable for lattices of finite height (as was the case for the MOP solution)
- We have to reconsider the MFP solution and avoid taking too many invalid paths into account
- Encode information about the paths taken into data flow properties themselves
- Introduce context information

MFP Counterpart

Context sensitive analysis: add context information

- call strings:
  - an abstraction of the sequences of procedure calls that have been performed so far
  - example: the program point where the call was initiated
- assumption sets:
  - an abstraction of the states in which previous calls have been performed
  - example: an abstraction of the actual parameters of the call

Context insensitive analysis: take no context information into account.

Call Strings as Context

- Encode the path taken
- Only record flows of the form (ℓ_c, ℓ_n) corresponding to a pro
call
- we take as context $\Delta =$ Lab* where the most recent label ℓ
procedure call is at the right end
- Elements of Δ are called call strings
- The sequence of labels $\ell_c^1, \ell_c^2, \ldots, \ell_c^n$ is the call string leading
current call which happened at ℓ_c^1; the previous calls where
$\ell_c^2 \ldots \ell_c^n$. If $n = 0$ then no calls have been performed so far.

For the example program the following call strings are of interes

$\Lambda, [11], [11, 5], [11, 7], [11, 5, 5], [11, 5, 7].[11, 7, 5], [11, 7, 7], \ldots$

Abstracting Call Strings

Problem: call strings can be arbitrarily long (recursive calls)

Solution: truncate the call strings to have length of at most k for some fixed number k

- $\Delta =$ Lab$^{\leq k}$
- $k = 0$: context insensitive analysis
  - Λ (the call string is the empty string)
- $k = 1$: remember the last procedure call
  - $\Lambda, [11], [5], [7]$
- $k = 2$: remember the last two procedure calls
  - $\Lambda, [11], [11, 5], [11, 7], [5, 5], [5, 7], [7, 5], [7, 7]$

References

- Material for this 4th lecture (part 2)

  `www.complang.tuwien.ac.at/markus/optub.html`
- Book

  Flemming Nielson, Hanne Riis Nielson, Chris Hankin:

  Principles of Program Analysis.

  Springer, (450 pages, ISBN 3-540-65410-0), 1999.
  - Chapter 2 (Data Flow Analysis)