Traces of control-flow graphs

a feasibility study of using trace theory for compilation

Simone Campanoni and Stefano Crespi Reghizzi
Politecnico di Milano, Dipartimento Elettronica e Informazione - DEI
stefano.crespireghizzi@polimi.it,
simone.campanoni@polimi.it

October 21, 2008

Outline

- Instruction rescheduling, as by compilers, formalized with trace theory
- Control-dependences, unlike data-dependences, cannot be arbitrarily assigned, but are determined by Control Flow Graph CFG
- CFG modelled as a local DFA
- Several properties are proved for control-dependences and traces (star-connection, unambiguity, unicity of C-dep.)
- Program transformation to max. parallel form obtained by data-flow equations
- Roadmap for studying important program transformations using trace theory

CFG and control-dependences

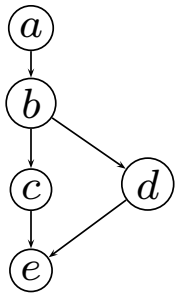
> from [J. Ferrante, K.Ottenstein, and J.Warren.]: The set of dependences for a program may be viewed as inducing a partial ordering on the statements and predicates in the program that must be followed to preserve the semantics of the original program. Dependences arise as the result of two separate effects. First, a dependence exists between two statements whenever a variable appearing in one statement may have an incorrect value if the two statements are reversed. . . . Second, a dependence exists between a statement and the predicate whose value immediately controls the execution of the statement.

- Data-dep. well studied in trace theory
- Control-dep. have ever been considered?
- Runs obtained by permuting independent instructions may not be compatible with Control-dep.

Valid yet inconsistent commutation

a : read (x, y);
b : if $x > 0$ goto c else goto d;
$c : x = x - 1$; goto e; $d : x = x + 1$; goto e;
e : print (y)

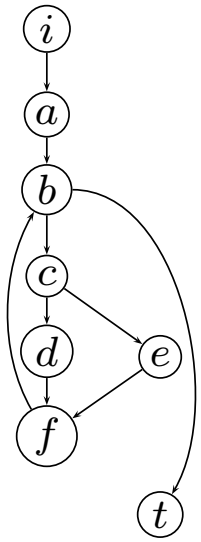
CFA



- Possible runs (or paths): $\{abce, abde\}$
- e data-dependent only on a, c and e independent
- $\{abec, abde\}$ represents same trace language but is not local lang.
- path $abdec$ violates program semantics since both successors of predicate b are executed!

CFG and Control-Flow Automaton

CFA



i	entry
a	statement
b	while condition
c	if condition
d	then statement
e	else statement
f	statement end
t	exit

- CFA is a local [Berstel and Pin] DFA $A = (Q, \Sigma, \delta, q_0, F)$ where: state set $Q = \Sigma$, initial/final states $i, \{t\}$, graph δ has one/two successors and no self-loops

- Corresponding language family $\mathcal{CFG}$ termed CFG family

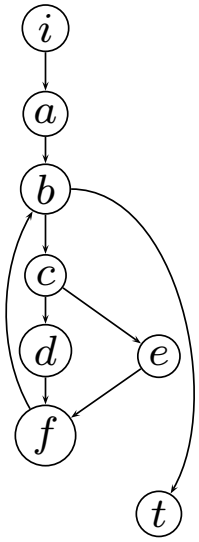
Post-dominance for CFA A

strict postdominance rel. $\vdash_s \subseteq \Sigma \times \Sigma$:

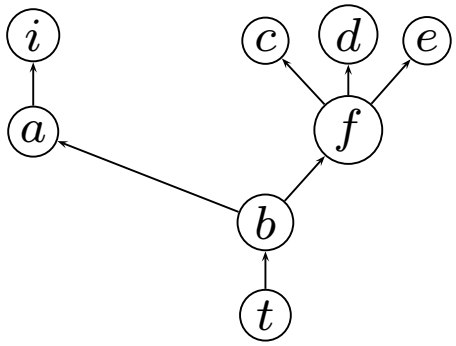
$$a \vdash_s b \Leftrightarrow \forall x \in L(A) : \underbrace{\pi_{\{a,b\}}(x)}_{projection} \in \{(a,b)^* a \cup \epsilon\}$$

postdominance rel. $\vdash$ adds identity to $\vdash_s$

CFA



postdominance tree $\vdash$



Instruction b postdominates instruction a if, and only if, after executing b, instruction a is always executed.

Control-dep. for CFG language L (or CFA A)

$\Sigma_2 \subset \Sigma$: letters (predicates) with 2 successors.

ternary control-dep. rel. $D_3 \subset \Sigma \times \Sigma_2 \times \Sigma$, a is control dependent on b via c: $(a,b)_c \in D_3(L) \Leftrightarrow (a \vdash c) \wedge \neg(a \vdash_s b) \wedge c \in suc(b)$

binary intermediate rel. $D_{3_2}(L) = \{(a,b) \mid (a,b)_c \in D_3(L)$ for some $c \in \Sigma\}$

binary control-dep. rel. $(a,b) \in D_2(L)$ iff $(a,b) \in D_{3_2}(L) \vee (b,a) \in D_{3_2}(L) \vee (a = b)\}$

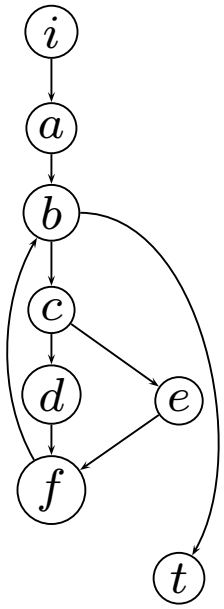
a is control-dep. on b via c iff b has a successor c s.t. if c is executed then a is surely executed, but, if the other successor of b is taken, it is not sure that a will be later executed.

D_{3_2} does not say which successor of the predicate is always followed by a.

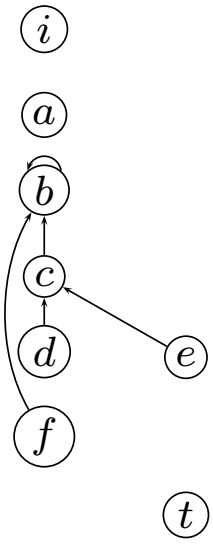
D_2 is symmetric and reflexive, to be consistent with trace theory.

Example

CFA



control dep. D_{3_2}



D_3	D_2
$(b,b)_c$	(b,b)
$(c,b)_c$	$(c,b),(b,c)$
$(d,c)_d$	$(d,c),(c,d)$
$(e,c)_e$	$(e,c),(c,e)$
$(f,b)_c$	$(f,b),(b,f)$
	$(i,i),(a,a),(c,c),$
	$(d,d),(e,e),(f,f),(t,t)$

Properties of control-dep.
Let L be a CFG language.

1. Control-dep. relation $D_3(L)$ is empty iff, for every letter $a \in \Sigma$, $suc_L(a) \neq 2$.

2. Let b be a predicate, with $suc_L(b) = \{a, d\}$. Then $(a, b)_a \in D_3(L)$ or $(d, b)_d \in D_3(L)$

3. For a CFA A with $L = L(A)$. The binary intermediate relation $D_{3_2}(L)$ contains a cycle iff the graph of A contains a circuit. Moreover, when a CFA circuit has more than one exit, the exit nodes exhibit mutual control-dependences.

Note for 3.: informally stated in [Ferrante, Ottenstein Warren]

Traces of CFG languages

Let L_0 be a CFG language.

1. The trace language $[L_0]_{D_2(L_0)}$ is called the trace language C-defined by L_0

2. The family of CFG trace languages $\mathcal{T}_{CFG}$ contains all T such that T is a trace language C-defined by L_0 for some $L_0 \in \mathcal{CFG}$

Inclusion within Recognizable trace lang.

For any CFG language, the graph of the binary control-dependence is connected for any iterative factor.

Lemma: star connection. Let $D_{3_2} = D_{3_2}(L)$ be the intermediate binary control-dep. rel.. Let w be a simple iterative factor (i.e., a simple circuit) and let $W = alph(w)$. Then for every pair of letters $a, e \in W$ such that e is an exit from w, there is in graph $D_{3_2|W}$ a (not necessarily directed) path, termed a D-path, connecting a to e.

Theorem: The family of CFG trace languages $\mathcal{T}_{CFG}$ is strictly included within the family $\mathcal{R}ec$ of recognizable trace languages.

Hint of proof. A trace language T is recognizable iff there exists a regular language X s.t every iterative factor of X is D-connected and $T = [X]$. Strict inclusion: $R = \{abc\}$ with dep. rel. $D = \{(a, b)\}$ is recognizable, but D differs from control rel. $D_2(R) = \emptyset$

Control equivalent automata (to model semantic-equivalent programs)

Def. Let $T_0 = [L_0]_{D_2(L_0)}$ be the trace lang. C-defined by L_0. The fam. of CFG lang. L over Σ , s.t. $D_3(L) = D_3(L_0)$ and $[L]_{D_2(L_0)} = T_0$, is $\mathcal{L}_C(L_0)$. Languages in this family are termed C-equivalent.

Theorem. The lang. family $\mathcal{L}_C(L_0)$ is strictly included within the family $\{R \mid R$ is a reg. lang. $\wedge\, [R]_{D_3(L_0)} = T_0\}$.

Weak inclusion is obvious. Strictness follows from the introductory ex. The set $L_0 = \{abce, abde\}$ has $D_2(L_0) = \{(b, c), (b, d)\}$ (omitting identity). Lang. $R = \{aebc, abde\}$ defines modulo $D_2(L_0)$ the same trace language C-defined by L_0, but it is not local.

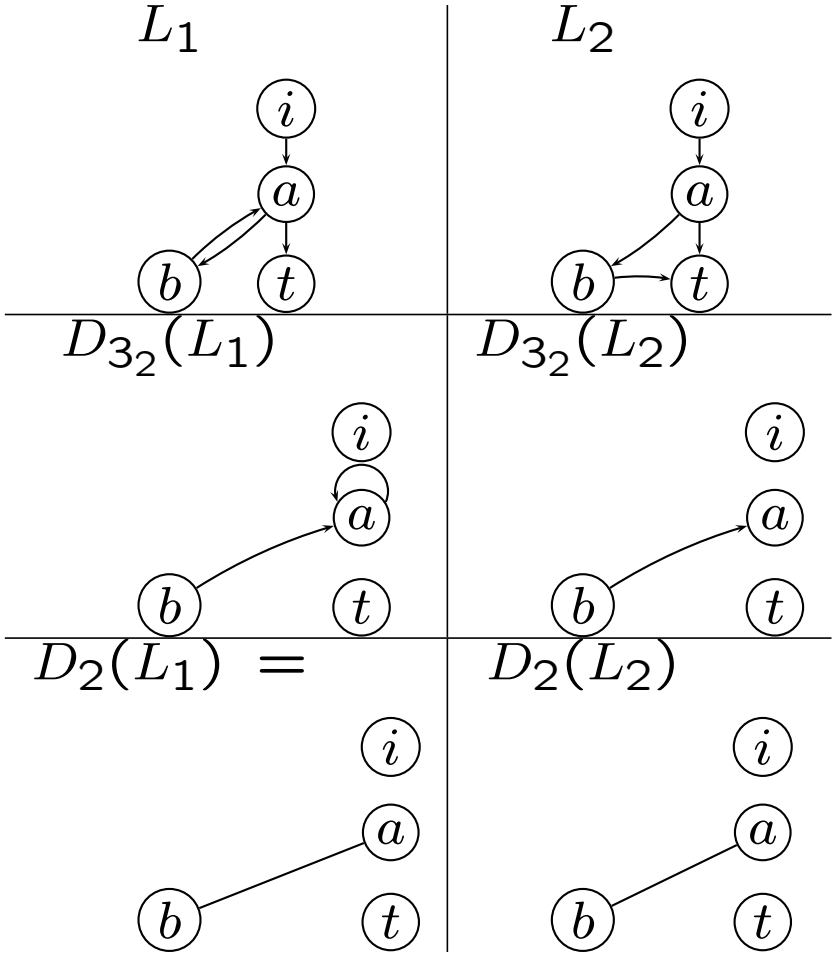
Traces determine control-dependences

Let L' and L'' be CFG languages over the same Σ and Σ_2.

Theorem. If the trace lang. C-defined by the two lang. are equal, then the binary control-dep. relations are equal:
$[L']_{D_2(L')} = [L'']_{D_2(L'')} \Longrightarrow D_2(L') = D_2(L'')$

Converse is false. Example in next slide.

Example of non-trace equivalent languages with same binary control-dependences



The converse of the theor, is likely to be true if one considers the more informative control-dep. relation D_3 instead of D_2

Unambiguity

- a rational trace lang. T over monoid $M(\Sigma, D)$ is unambiguous if $\exists$ a regular lang. X s.t. $T = [X]_D$ and $\forall$ trace $t \in T$, X contains exactly one representative for t

- then lang. X unambiguously defines trace lang. T

Theorem. Consider a CFG lang. L_0, the C-defined trace lang. T_0, and the family $\mathcal{L}_C(L_0)$ of CFG lang. which C-define T_0. Every lang. in this family unambiguously defines T_0.

The proof by contradiction is rather complex. it consists in showing that for any distinct strings $w_1, w_2 \in L_0$ it is $[w_1]_{D_2(L_0)} \neq [w_2]_{D_2(L_0)}$

Transformation for program parallelization: ordering by degree of parallelism

1. Binary relation $\geq_P$ over C-equivalent string languages.

2. For a string $s \in \Sigma^*$, clique decomposition $s = s_1 s_2 ... s_n$; each s_i is an independence clique, and $\forall$ letter in s_j, $1 \leq j < n$, $\exists$ a dependent letter in s_{j+1}

3. Let $[L_1]_{D_2(L_1)} = [L_2]_{D_2(L_2)}$ hence from Theor.: $D_2(L_1) = D_2(L_2)$

4. Partial order $\geq_P$ on strings $s_1 \in L_1, s_2 \in L_2$ representing the same trace. s_1 is more parallel than s_2, $s_1 \geq_P s_2$, if, for clique decompositions $s_1 = s_{1,1} s_{1,2} ... s_{1,n}$ and $s_2 = s_{2,1} s_{2,2} ... s_{2,m}$ it is $n < m$ or, if $n = m$ then, the differences of cardinalities, $|s_{1,i}| - |s_{2,i}|$, $1 \leq i < n$, form a sequence with signs in pp^*n^* where $p = +$ and $n \in \{-, 0\}$.

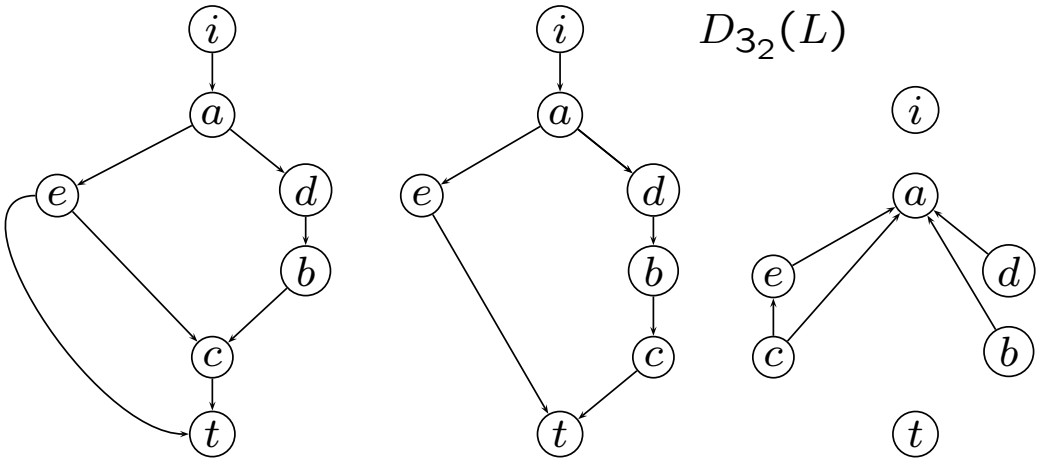
5. Lang. L_1 is more parallel than L_2, $L_1 \geq_P L_2$, iff $\forall$ string $s_1 \in L_1$ and $s_2 \in L_2$ with $[s_1]_{D_2(L_1)} = [s_2]_{D_2(L_1)}$, it is $s_1 \geq_P s_2$.

Data-flow equations for transforming automaton 1

1. L be in $\mathcal{CFG}$, consider the pre-dominance $\dashv$ and post-dominance $\vdash$ relations

2. define the reflexive partial order $\preceq$ over Σ
   $a \preceq b$ iff $(a \dashv b) \vee (b \vdash a)$
   (i.e. a necessarily precedes b or is necessarily followed by b)
   The non-reflexive relation is $a \prec b$

$L \in \mathcal{CFG}$ $\preceq$ *of* L $D_{3_2}(L)$



Data-flow eq. for transforming automaton. 2

1. Data-flow equations in the style of static program analysis. The result tells, for each node n of aut. A, which letters can be moved into it.

2. Let $n, b, p \in \Sigma$:

$$in_b[n] = \bigcap_{p \in predec(n)} out_b[p]$$

$$out_b[n] = (in_b[n] - kill_b[n]) \cup gen_b[n]$$

Unknowns are: $in_b[n], out_b[n]$
Constant sets are:

3. $$\begin{cases} gen_b[a] = \emptyset, & \text{if } a \neq b; \\ gen_a[a] = \{e | a \preceq e\}, & \text{otherwise.} \end{cases}$$

4. $$\begin{array}{ll} kill_b[a] = \Sigma \setminus \{e \mid a \prec e\}, \text{if } (b,a) \in D_{3_2}(L) \wedge & a \neq b; \\ kill_b[a] = \{e \mid \neg((a \preceq e) \vee (e \preceq a))\}, & \text{if } a \preceq b \text{ ;} \\ kill_b[a] = \emptyset, & \text{otherwise.} \end{array}$$

Data-flow eq. solution

	i	a	b	c	d	e	t
$in[i]$	Σ	Σ	Σ	Σ	Σ	Σ	Σ
$in[a]$	Σ	Σ	Σ	Σ	Σ	Σ	Σ
$in[b]$	i, a, b, c, d, t	i, a, b, c, d, t	b, c, d, t	b, c, d, t	b, c, d, t	b, c, d, e, t	b, c, t
$in[c]$	i, a, t	i, a, t	b, c, d, t	t	b, c, d, t	e, t	t
$in[d]$	Σ	Σ	b, c, d, e, t	b, c, d, e, t	b, c, d, e, t	b, c, d, e, t	b, c, d, e, t
$in[e]$	Σ	Σ	b, c, d, e, t	b, c, d, e, t	b, c, d, e, t	b, c, d, e, t	b, c, d, e, t
$in[t]$	i, a, t	i, a, t	b, c, d, t	t	b, c, d, t	e, t	t

	i	a	b	c	d	e	t
$out[i]$	Σ	Σ	Σ	Σ	Σ	Σ	Σ
$out[a]$	Σ	Σ	b, c, d, e, t	b, c, d, e, t	b, c, d, e, t	b, c, d, e, t	b, c, d, e, t
$out[b]$	i, a, b, c, d, t	i, a, b, c, d, t	b, c, d, t	b, c, d, t	b, c, d, t	b, c, d, e, t	c, t
$out[c]$	i, a, t	i, a, t	b, c, d, t	c, t	b, c, d, t	e, t	t
$out[d]$	i, a, b, c, d, t	i, a, b, c, d, t	b, c, d, t	b, c, d, t	b, c, d, t	b, c, d, e, t	b, c, t
$out[e]$	i, a, e, t	i, a, e, t	b, c, d, e, t	e, t	b, c, d, e, t	e, t	t
$out[t]$	i, a, t	i, a, t	b, c, d, t	t	b, c, d, t	e, t	t

Construction of Maximally Parallel Automaton A_{max} control-equivalent to A

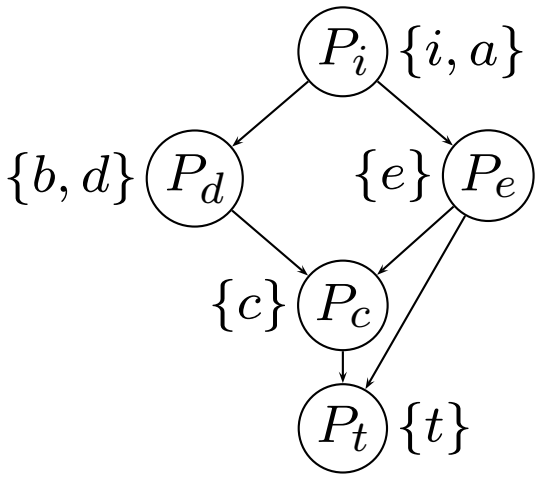
$A_{max} = (\text{states} = Q_{max}, \Sigma_{max}, \delta_{max}, P_i, P_t)$ where

$Q_{max} = \Sigma_{max} = \{P_b \mid P_b \text{ indep. clique }\}$

$P_b = \{a \in \Sigma \mid b \in out_a[a] \wedge \forall c \in out_a[a]\ (b \preceq c)\}$
Each letter b occurs in only one clique of Q_{max}

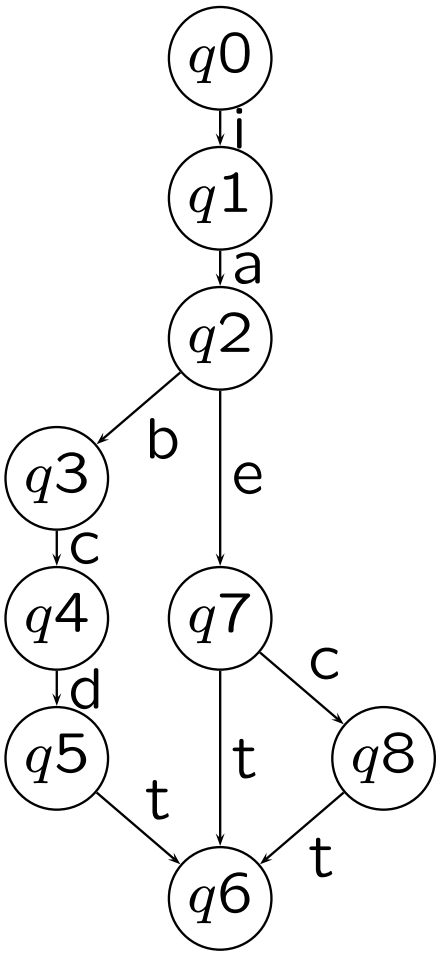
Transition function: for letters $x \neq y$
$\delta_{max}(P_x, P_y) = P_y$ iff, for letters $a \in P_x, b \in P_y$ it is $\delta(a, b) = b$

Maximally parallel program A_{max} for $L = L(A)$



It has same control-dep. as A.
Each P_i is an I-clique; its letters can be executed in parallel.

Foata Normal Form is not C-equivalent to A



Not a local automaton: instruction c is replicated. (Compilers replicate code, but such transformations are not considered here.)

Conclusion: done or easy to do.

1. simple program transformations (instruction rescheduling) are conveniently represented by partial commutations, using the control-dependence relation, which depends on the given program, not just on the properties of individual instructions.

2. data-dependences (RAW, etc.) should be also considered. It should be an easy job to superimpose data- onto control-constraints.

3. it should be feasible to write data-flow eq. for a achieve uniform parallelism at all steps, instead of anticipating instructions as we did in the maximally parallel example.

Conclusion: a manifesto for transferring trace theory to program transformation methodology.
This attempt at expressing program transformations via (suitably enriched) trace theory should be continued to cover more important transformations:

1. Loop-invariant extraction. When a loop invariant(e.g. a constant assignment x:=37) is moved from the loop body to its preface, the new CFA is not commutatively equivalent to the original, yet the two programs are semantically equivalent. To model this, invariant statements can be qualified as idempotent.

2. Speculative execution: moving a control-dependent instruction across the condition

it depends on. Such transformation requires an undo instruction, to be modeled via a partially inverse monoid (as suggested by Diekert and Silva)

3. Transactional memory; this HW mechanism for speculative multi-threading is similar, but undoing is applied to a series of instructions and needs a chain of undo operations.

Last, but not least, software pipelining and other loop parallelizing transformations: here it is less clear what extensions to trace theory would be required. One possibility is to consider the alphabet to be enumerable but not finite, in order to differentiate the instructions at different loop iterations.

We would be happy to develop, implement and experiment trace-based compiler design tools, as we have developed and are supporting a free-software compilation platform (ILDJIT) which is suitable for such experimentation.
We are looking around for fitting theoretical models and for algorithms!