Traces of Control-Flow Graphs*

Simone Campanoni** and Stefano Crespi Reghizzi***

Politecnico di Milano, Dipartimento Elettronica e Informazione - DEI
P.za Leonardo da Vinci 32, I–20133, Milano
simone.campanoni@mail.polimi.it, stefano.crespireghizzi@polimi.it

Abstract. This is a new applied development of trace theory to compilation. Trace theory allows to commute independent program instructions, but overlooks the differences between control and data dependencies. Control(C)-dependences, unlike data-dependences, are determined by the Control Flow Graph, modelled as a local DFA. To ensure semantic equivalence, partial commutation must preserve C-dependences. New properties are proved for C-dependences and corresponding traces. Any local language is star-connected with respect to C-dependences, hence this trace language family is recognizable. Local languages unambiguously represent traces. Within the family of local languages with the same C-dependences, we construct the language such that instructions are maximally anticipated. This language differs from the Foata-Cartier normal form. Future directions for application of trace theory to program optimization are outlined.

1 Introduction

This is a research on the application of trace theory to compilation. Optimizing compilers transform a program in many ways. Transformations which only change the order of execution of instructions are known as rescheduling. Inside compilers, the Control Flow Graph CFG representation carries the essential information needed for performing program transformations. CFG nodes represent the instructions (such as assignments and predicates), and the arcs the predecessor-successor relation. A CFG is conveniently viewed as a Deterministic Finite Automaton (DFA) of the local [3] type. Program instructions interfere in different ways, termed *dependences.*

> 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 [Data-]dependence exists between two statements whenever a variable

* Preliminary presentation at ESF-AutoMathA Workshop on Developments and New Tracks in Trace Theory, Cremona 2008.

** Partially supported by STMicroelectronics.

*** Partially supported by COFIN "Linguaggi Formali" and CNR-IEIIT.

V. Diekert and D. Nowotka (Eds.): DLT 2009, LNCS 5583, pp. 156–169, 2009.
© Springer-Verlag Berlin Heidelberg 2009

> appearing in one statement may have an incorrect value if the two statements are reversed. ... Second, a [Control(C)-]dependence exists between a statement and the predicate whose value immediately controls the execution of the statement [7].

Mazurkiewicz's theory quite abstractly represents the program statements as letters of an alphabet, and D-dependences by means of a binary relation. This abstraction is realistic because a data-dependence, say of the Read-After-Write type, is solely determined by the variables used or defined in the two statements, and not by their position in the CFG. This allows to find a program (i.e. a model in the logical sense) for any arbitrarily given data-dependence relation over the alphabet.

On the other hand a C-dependence is a structural property of the program CFG. If C-dependences are arbitrarily assigned, discrepancies may arise between the real program and its idealization by means of trace theory. More precisely, a C-dependence is generally neither symmetric nor reflexive; and, for a given partially commutative alphabet and CFG, it may well be that no program exists with the corresponding C-dependence relation.

Related work. A few formal studies of C-dependences exist such as [10], but not oriented towards program transformation. We know of just one work [5] on locally defined traces, investigating the word problem for certain program loops. In looser sense, some similarity of objectives may be seen between our work and the much more established line [8] investigating the application of asynchronous automata to concurrent programs.

As we did not find any previous work useful to apply trace theory for program optimization, we investigated how C-dependences constrain commutation. The next simple example introduces the problem.

$$\begin{array}{c} a: \text{ read } (x, y); \\ b: \text{ if } x > 0 \text{ goto } c \text{ else goto } d; \\ c: x = x - 1; \text{ goto } e; \qquad d: x = x + 1; \text{ goto } e; \\ e: \text{ print } (y); \end{array}$$

The runs are just $\{abce, abde\}$. Since instruction e is data-dependent only on a, among the runs obtained by permuting independent instructions such as c and e, we have $\{abec, abde\}$. Clearly the two sets represent the same trace language. But strings of the latter set, when viewed as CFG paths, imply the existence of path $abdec$, which violates the program semantics since both successors of predicate b are executed! The inconsistency comes from overlooking an essential constraint: instruction rescheduling must ensure that the original and the transformed program have the same C-dependences.

This paper sets a new rigorous framework for such program transformations, combining and extending some basic results of trace and compilation theories. The paper proceeds as follows. Sect. 2 contains basic definitions from compiler theory, and states simple but elsewhere non-available properties of post-dominance and C-dependence relations. Sect. 3 specializes trace theory for C-dependences. Any local language is star-connected with respect to

C-dependences, hence this trace language family is recognizable. Moreover CFG local languages unambiguously represent traces, and identity of traces implies identity of C-dependences. Sect. 4 is a significant application: within the family of local languages with the same C-dependences, we define and construct the language such that instructions are maximally anticipated. This language differs from the Foata-Cartier normal form, which in general is not a local language. Sect. 5 lists future directions for application of trace theory to program parallelization.

2 Basic Definitions and Properties

For the basic concepts of formal language and trace theory we refer primarily to [6], for compiler theory to e.g. [2,1,9]. Let Σ be a finite alphabet. The set of all strings over Σ is denoted by Σ^*, including the empty string, denoted by ε. For any string $x \in \Sigma^*$, $|x|$ denotes the length of x, $x(i)$, with $1 \leq i \leq |x|$, is the i-th character of x, $alph(x)$ denotes the set of letters present in x, and $\pi_\Delta(x)$ denotes the projection of x on a set $\Delta \subseteq \Sigma$ of letters.

In compilation and software engineering a program is often represented by a control flow graph CFG, a single entry (i), single exit (t) directed graph $G = (\Sigma, E)$. Let $pred(b), succ(b)$ denote the predecessors and successors of node b. The following customary hypotheses will be tacitly assumed.

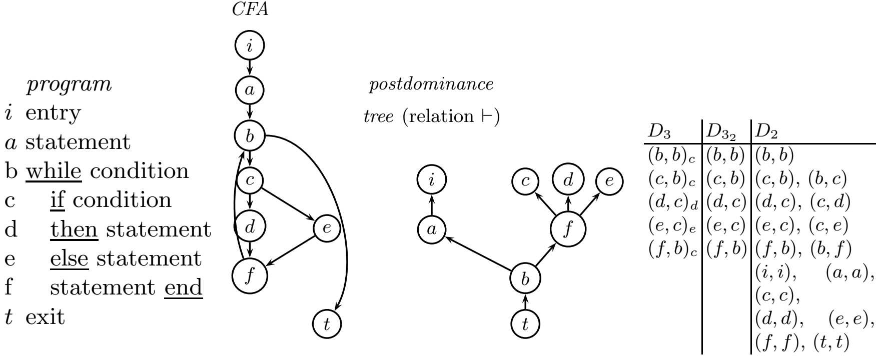
1. Each node has at most two successors. A node with two successors is a *predicate* (or conditional instruction).
2. The successor of a node cannot be the node itself, i.e. the CFG has no self-loops.
3. For any node b there exists a path from the i to b, and a path from node b to t.

The definition of CFG is next rephrased within automata theory, using a restricted type of DFA known as *local automaton* [3].

Definition 1. *A Control Flow Automaton representing a CFG* $G = (\Sigma, E)$ *is a DFA* $A = (Q, \Sigma, \delta, q_0, F)$ *where the state set is* $Q = \Sigma$*, the initial and final states are* $q_0 = i$*,* $F = \{t\}$ *, the graph of the transition function* δ *is such that* $\delta(p, b) = q$ *if, and only if,* $p = a$*,* $q = b$ *and* $a \rightarrow b \in E$*. A recognizes a language* $L(A)$*. The corresponding language family is termed CFG family and denoted by* $\mathcal{CFG}$*.*

A CFA defines a local language [3], and $\mathcal{CFG}$ is strictly included in the family of local regular languages, because of the above restrictions on CFG. In particular, a language in $\mathcal{CFG}$ is never empty and may not contain the empty string.

It is known that local automata are convenient visualized identifying the states with terminals and omitting the transitions labels, as shown in figure 1.



D_3	D_{3_2}	D_2
$(b,b)_c$	(b,b)	(b,b)
$(c,b)_c$	(c,b)	(c,b), (b,c)
$(d,c)_d$	(d,c)	(d,c), (c,d)
$(e,c)_e$	(e,c)	(e,c), (c,e)
$(f,b)_c$	(f,b)	(f,b), (b,f)
		(i,i), (a,a),
		(c,c),
		(d,d), (e,e),
		(f,f), (t,t)

Fig. 1. Program, CFA, strict post-dominance tree, and C-dependence relations

In compiler theory two properties of a CFG play a major role: predominance and postdominance. An instruction a post-dominates instruction b if, and only if, after executing instruction b, instruction a is always executed.

Definition 2. *Let A be a CFA, and let $a \neq b$. The strict postdominance relation $\vdash_s \subseteq \Sigma \times \Sigma$ is*

$$a \vdash_s b \text{ if, and only if, for any } x \in L(A): \ \pi_{\{a,b\}}(x) \in \{(a,b)^* a \cup \epsilon\}$$

Then the postdominance relation $\vdash$ is obtained adding to $\vdash_s$ the identity relation. The immediate postdominance relation $\vdash_i$ is

$$a \vdash_i b \text{ iff } a \vdash_s b \text{ and } a \text{ does not postdominate any other dominator of } b$$

Similarly, instruction a strictly *predominates* instruction b if, and only if, before executing instruction b, instruction a is always executed. The notation for predominance is $a \dashv_s b$. Postdominance and predominance are partial order reflexive relations and more precisely tree partial orders.

The concept of an instruction depending on a predicate has been formalized within compiler and software engineering research. Slightly different formulations exist (e.g. [10]) and we follow [7,2,9]).

For a CFA A or $\mathcal{CFG}$ language L, let the sub-alphabet $\Sigma_2 \subset \Sigma$ contain the letters having two successors, and let $\Sigma_1 = \Sigma \setminus \Sigma_2$. The letters in Σ_2 represent predicates.

Definition 3. *For a $\mathcal{CFG}$ language $L \subseteq \Sigma^+$ the ternary C-dependence relation $D_3 \subset \Sigma \times \Sigma_2 \times \Sigma$, pronounced as a is control dependent on b via c, is defined as*

$$(a,b)_c \in D_3(L) \text{ if, and only if, } (a \vdash c) \wedge \neg(a \vdash_s b) \wedge c \in succ(b).$$

By erasing the third argument, we obtain the binary intermediate relation

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

Then closing $D_{3_2}(L)$ by means of commutation and adding the identity relation, we obtain the binary C-dependence relation

$$(a, b) \in D_2(L) = \{(a, b) \mid (a, b) \in D_{3_2}(L) \vee (b, a) \in D_{3_2}(L) \vee (a = b)\}$$

Examples are shown in Figures 1 and 2. Notice that in the ternary relation a may coincide with c or a may coincide with b, but b necessarily differs from c because the successor relation is irreflexive. Intuitively, a is control dependent on b via c when predicate b has a successor c such that, if c is executed then also a is surely executed, but, if the other successor of b is taken, it is not certain that a will be later executed. The intermediate binary relation is less informative; it does not say which of the successors of the predicate is always followed by a. Finally the binary relation is symmetric and reflexive, to be consistent with the usual assumptions of trace theory.

Some properties of C-dependences are next stated, which are later needed. Let A be a CFA and $L \subseteq \Sigma^*$ be its language.

Statement 1. *The C-dependence relation $D_3(L)$ is empty iff $\Sigma_2 = \emptyset$.*

In other words, there no C-dependences if, and only if, the CFA graph has no bifurcation. Notice that the "only if" part of the statement is not obvious.

Statement 2. *Let $b \in \Sigma_2$, with $succ(b) = \{a, d\}$. Then at least one of $(a, b)_a$ or $(d, b)_d$ is in $D_3(L)$.*

Proof. By contradiction, assume

$$\exists a \in \Sigma \text{ such that } |succ(a)| > 1 \text{ and } \forall b \in succ(a) \text{ it is } (b, a)_b \notin D_3(L).$$

Since by definition of CFA, $\forall b \in succ(a)$ it is $b \neq a$, it follows that $\forall b \in succ(a)$, b postdominates a. Let now $\{b_1, b_2\} = succ(a)$; but if $b_1 \vdash a \wedge b_2 \vdash a$, then either $b_1 \vdash b_2$ or $b_2 \vdash b_1$. Considering the case $b_1 \vdash b_2$ (the other case is analogous), any string $w \in L$ containing b_2 can be factorized as $w = y_1 b_2 y_2 b_1 y_3 t$, with b_2 not occurring in y_2 or in y_3. But since $b_2 \in succ(a)$, there exists a string $w_1 \in L$ such that $w_1 = x_1 a b_2 x_2 t$, with b_1 not occurring in x_2. Hence b_1 does not postdominate a, a contradiction. Moreover, Figure 1 provides an example where one successor (t) of a predicate (b) is not control dependent on the predicate. □

Thus (at least) one of the successors of a predicate is control dependent on it.

For a CFA, a *circuit* $O = a_1, a_2, \ldots, a_n$ is a sequence of states $a_i \in \Sigma$, such that for all positions but the last, a_{i+1} is in $succ(a_i)$ and a_1 is in $succ(a_n)$. A node a_i present in O, having a successor $b \in succ(a_i)$ not in O, is called an *exit from circuit O*. An *iterative factor* of a language L is a string w such that $xw^*z \subseteq L$, for some strings x and z. A circuit or iterative factor is *simple* if all letters are distinct. Clearly for a $\mathcal{CFG}$ language a simple iterative factor is a simple circuit of the CFA graph.

The next statement, informally presented in [7], characterizes the cases of cyclicity of the C-dependence relation. The property immediately follows from Lemma 1 of Section 3.

Statement 3. *The binary intermediate relation $D_{3_2}(L)$ contains a cycle if, and only if, the graph of A contains a circuit. Moreover, when a CFA circuit has more than one exit, the exit nodes exhibit mutual C-dependences.*

For instance, consider a program loop such as the one in fig.1. Node b is the the loop exit, and it is control dependent on itself. Notice that this statement does not imply that a set of letters that make a loop within the graph of A, make a loop within D_{3_2} as well.

The next statement relates C-dependence and pre-, post-dominance relations.

Statement 4. *Let $a, b \in \Sigma$. If $\nexists c, d \in \Sigma$ such that $(a, c) \in D_{3_2}(L)$ and $(b, d) \in D_{3_2}(L)$, then the following two conditions are met: $a \dashv b \vee b \dashv a$ and $a \vdash b \vee b \vdash a$.*

3 Control Flow and Traces

This central section studies traces and C-dependences.

Let $D \subseteq \Sigma \times \Sigma$ be a symmetric and reflexive *dependence* relation, and its complement be denoted by I and named the *independence* relation. The dependence alphabet is the pair (Σ, D). The equivalence relation over Σ^* induced by I is denoted by $\sim_I$. The free partially commutative monoid, i.e. the quotient of Σ^* by the congruence $\sim_I$ is denoted by $M(\Sigma, I)$. For a string $x \in \Sigma^*$, the equivalence class of x under $\sim_I$ is called a *trace* and denoted by $[x]_I$. The mapping from strings to traces is denoted by φ: $\varphi(x) = [x]_I$. For a string language $X \subseteq \Sigma^*$, the mapping $\varphi(L)) = \{[x]_I \mid x \in X\}$ produces a *trace language*, also denoted by $[X]_I = \varphi(L)$. We use interchangeably I and its complement D, if no confusion arises. The subscript I is dropped if the (in)dependence relation is understood.

A trace or string is *D-connected* if its letters induce a connected subgraph of the dependence graph; a language is D-connected if every sentence is so.

Given a trace language T over a trace monoid $M(\Sigma, D)$, the family of all languages $L \subseteq \Sigma^*$ such that $[L]_I = T$ is denoted by $\mathcal{L}(T)$. Any language in $\mathcal{L}(T)$ is a *representative* of T.

For a subset $\Delta \subseteq \Sigma$ and a binary relation $D \subseteq \Sigma \times \Sigma$, the relation (or graph) *induced by* Δ is $D_{|\Delta} = D \cap \Delta^2$.

We refer to [6,4] for the notions of recognizable $\mathcal{R}ec$, rational $\mathcal{R}at$, and unambiguous trace language, and for the Foata-Cartier Normal Form.

Traces of $\mathcal{CFG}$ Languages

We consider a control-flow automaton A_0, recognizing the language $L_0 = L(A_0) \subseteq \Sigma^*$, which is, as we know, a local regular language and more specifically a member of the $\mathcal{CFG}$ language family. Let $D_3(A_0) = D_3(L_0)$ be the C-dependence relation and D_2 its symmetric and reflexive embedding. Since we do not consider data-dependences (except in Sect. 5), we may take the independence relation to be $I = (\Sigma \times \Sigma) \setminus D_2$.

A regular expression is *D-star-connected*, if the star is used over D-connected languages only. For a CFA, the definition becomes: if, for every circuit of the DFA graph, the state labels are D_2-connected.

Definition 4. *Let L_0 be a language in $\mathcal{CFG}$. The trace language $[L_0]_{D_2(L_0)}$ is called the* trace language C-defined *by L_0. The family of CFG trace languages is*

$$\mathcal{T}_{CFG} = \{T \mid T \text{ is a trace language C-defined by } L_0, \text{ for some } L_0 \text{ in } \mathcal{CFG}\}$$

The next technical Lemma will be used to prove that, for any CFG language, the graph of the D_{3_2} relation is connected for every iterative factor.

Lemma 1. *Let $L \subseteq \Sigma^*$ be a language in $\mathcal{CFG}$ and $D_{3_2} = D_{3_2}(L)$ be the intermediate binary C-dependence relation. Let w be a simple iterative factor with $W = alph(w)$. Then for every pair of letters $a, e \in W$ such that e is an exit from w, the graph $D_{3_2\,|W}$ contains a (not necessarily directed) path, termed a D-path, between a and e.*

Proof. The proof is by induction. Let $PATHS_k$, $k \geq 2$, be the set of CFG paths of length k contained in circuit w, and denote with $EPATHS_k$ the paths ending in an exit node. We say $EPATHS_k$ has the D-connection property, if, for every path z in $EPATHS_k$, the graph $D_{3_2\,|alph(z)}$ is connected[1].

Induction base. For circuit w, let ab be a path in $EPATHS_2$. Since b exits from w, from the definition of control-dependence it follows that $(a, b) \in D_{3_2}$. Thus $EPATHS_2$ has the D-connection property.

Inductive hypothesis. From the induction base, we may assume that for circuit w with $|w| = k$, the set $EPATHS_j$ has the D-connection property, for all $2 \leq j \leq k - 1$.

Induction step. We prove that $EPATHS_k$ has the D-connection property. Let $b_1 b_2 \ldots b_{k-1} b_k$ be a path in $EPATHS_k$. Since the path $x = b_2 \ldots b_{k-1} b_k$ is in $EPATHS_{k-1}$, it has the D-connection property. It suffices to show that b_1 is D-connected to a letter in $alph(x)$, and we prove it for b_2. Two cases arise.

- b_2 **postdominates** b_1**.** Since b_2 is control-dependent on some letter in $\{b_2, \ldots, b_{k-1} b_k\}$, from the definition of control-dependence it follows b_1 is dependent on the same letter.
- b_2 **does not post-dominate** b_1**.** Then b_1 is a predicate with b_2 as one of its successors. From Statement 2 it follows that $(b_2, b_1)_{b_1} \in D_3$. □

Clearly the property remains true for non-simple iterative paths. The next trace language family inclusion follows immediately.

Theorem 1. *The $\mathcal{T}_{CFG}$ family is strictly included within the family $\mathcal{R}ec$ of recognizable trace languages.*

Proof. Inclusion follows from the fact [6] that a trace language T is recognizable, if, and only if, there exists a regular language $X \subseteq \Sigma^*$ such that every iterative factor of X is D-connected and $T = [X]$. Moreover, the inclusion is strict: the (finite) language $R = \{abc\}$ with the dependence relation $D = \{(a, b)\}$ is recognizable, but D differs from the binary control relation $D_2(R)$, which is empty (Statement 1). □

[1] We prefer to use the intermediate binary relation D_{3_2} instead of D_2 because it makes more perspicuous the arguments based on the properties of control-dependences.

Control Equivalent Automata

We have argued in the introduction that a CFG language, though equivalent modulo the C-independence relation to a given language, may not correspond to a semantically equivalent program. This fact is precisely stated in the next theorem.

Theorem 2. *Let L' and L'' be languages in $\mathcal{CFG}$ over the same ranked alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$. If the trace languages C-defined by the two languages are identical, then the binary C-dependence relations are equal, in formula*

$$\text{if } [L']_{D_2(L')} = [L'']_{D_2(L'')} \text{ then } D_2(L') = D_2(L'')$$

but the converse implication does not hold.

In other words, for a given partition of the alphabet into predicates (Σ_2) and non-predicates, a trace language uniquely determines the C-dependence relation for all CFG representatives. On the other hand, two CFG languages having equal D_2, may represent different trace languages and therefore, they are not semantically equivalent when viewed as programs. For this reason, the theorem calls for a new stronger concept of equivalence.

Definition 5. *Let $L_0 = L(A_0)$ be a language in $\mathcal{CFG}$ and let $T_0 = [L_0]_{D_2(L_0)}$ be the C-defined trace language. The family $\mathcal{L}_C(L_0) \subseteq \mathcal{CFG}$ includes the languages L, such that $D_3(L) = D_3(L_0)$ and $[L]_{D_2(L_0)} = T_0$. The corresponding family of CFG automata is denoted by $\mathcal{A}_C(L_0)$. The languages (or automata) in these families are termed C-equivalent.*

For a given CFG language, we are interested in studying C-equivalent languages, having desirable properties. With the terminology of compilers, they can be viewed as obtained by legal program transformations.

To conclude this part, we formalize a fact anticipated in the introduction: a regular language, which represents the same trace as a CFG language L_0, may not even be in $\mathcal{CFG}$.

Statement 5. *Let L_0 be a language in $\mathcal{CFG}$ and T_0 its C-defined trace language. The language family $\mathcal{L}_C(L_0)$ is strictly included within the family $\{R \mid R \text{ is a regular language and } [R]_{D_3(L_0)} = T_0\}$.*

Proof. Weak inclusion is obvious. Strictness of inclusion follows from the example in Sect. 1. The set $L_0 = \{abce, abde\}$ is in $\mathcal{CFG}$, with $D_2(L_0) = \{(b, c), (b, d)\}$ (omitting the identity relation). The language $R = \{aebc, abde\}$ defines modulo $D_2(L_0)$ the same trace language C-defined by L_0, but it is not a local language, hence not a CFG language. □

Unambiguity

A natural question to ask concerns ambiguity. We recall a rational trace language T is unambiguous if, there exists a regular language X such that $T = [X]_I$ and for

every trace $t \in T$, X contains exactly one representative for t. In that case we say that language X *unambiguously defines* trace language T. Since all recognizable trace languages are unambiguous, from Theor. 1 the same holds for the $\mathcal{T}_{CFG}$ family. The next statement solves the question whether every $\mathcal{CFG}$ language unambiguously defines its trace language.

Theorem 3. *Consider a language L in $\mathcal{CFG}$, the C-defined trace language T, and the family $\mathcal{L}_C(L)$ of CFG languages which C-define T. Every language in this family unambiguously defines T.*

Proof. It is based on Lemma 2 and Lemma 3. We show that for any distinct strings $w_1, w_2 \in L$ it is $[w_1]_{D_2(L)} \neq [w_2]_{D_2(L)}$. Assume by contradiction that $[w_1]_{D_2(L)} = [w_2]_{D_2(L)}$.

If $D_3(L) = \emptyset$, from statement 1 it follows that each letter has at most one successor, and, since the first letter must be i, $|L| = 1$, hence $w_1 = w_2$, a contradiction.

It remains to analyze the case $D_3(L) \neq \emptyset$. Consider the longest common prefix xa of the two strings $w_1 = xaby_{11} \qquad w_2 = xacy_{21}$ with $b \neq c$ and $y_{11}, y_{21} \in \Sigma^*$.

Since $[w_1]_{D_2(L)} = [w_2]_{D_2(L)}$ implies $\pi_b(w_1) = \pi_b(w_2) \wedge \pi_c(w_1) = \pi_c(w_2)$, letter b must occur in y_{21} and letter c must occur in y_{11}, hence the factorizations

$$w_1 = xaby_{12}cy_{13} \qquad w_2 = xacy_{22}by_{23}$$

where $y_{12}, y_{13}, y_{22}, y_{23} \in \Sigma^* \wedge c \notin y_{12} \wedge b \notin y_{22}$. Moreover, since w_1 and w_2 are in the same trace, b and c are independent letters (i.e. $(b, c) \notin D_{3_2}(L)$).

From Lemma 3, it follows that $\forall d \in y_{12}, (c, d) \notin D_{3_2}(L)$ and $\forall d \in y_{22}, (b, d) \notin D_{3_2}(L)$. Moreover, from Lemma 2, we have $c \vdash b$ and $b \vdash c$. Since the post-dominance relation makes a tree over Σ, and $c \vdash b$ and $b \vdash c$, then $c = b$. Contradiction found. □

Lemma 2. *Let $L \in \mathcal{CFG}$ contain a sentence of the form $xyaz$, for a letter a, and some strings $x, z \in \Sigma^*, y \in \Sigma^+$. Then*

$$\exists c, d \in ya \textit{ such that } (a, c)_d \in D_3 \textit{ if, and only if, } \exists e \in ya \textit{ such that } \neg(a \vdash e)$$

Proof. We provide the proof in two rounds.

$\underline{\exists c, d \in ya \mid (a, c)_d \in D_3 \Rightarrow \exists e \in ya \mid \neg(a \vdash e)}$ By contradiction, we suppose that:

$$(\exists c, d \in ya \mid (a, c)_d \in D_3) \wedge (\nexists e \in ya \mid \neg(a \vdash e))$$

Then it follows $\forall e \in ya, (a \vdash e)$. Since $(a, c)_d \in D_3$ follows that $\neg(a \vdash c)$ by definition of D_3. Since $c \in ya \wedge \forall e \in ya, (a \vdash e)$ then $a \vdash c$. Contradiction found.

$\underline{\exists c, d \in ya \mid (a, c)_d \in D_3 \Leftarrow \exists e \in ya \mid \neg(a \vdash e)}$ We provide the proof by induction.

Induction base. For the string ya, let the length of y be one and then $y \in \Sigma$. Since $a \vdash a$ by definition of the pre-dominance relation, then the Lemma becomes $\neg(a \vdash y) \Rightarrow (a, y)_a \in D_3$ which is true by definition of D_3.

Inductive hypothesis. From the induction base, we may assume that for any y with length equal to $n-1$ the Lemma holds.

Induction step. We prove that for any y with length n, the Lemma holds. We can write the string as xdy_1az where $y = dy_1$. If $\neg(a \vdash y_1(1))$, then by the Inductive hypothesis, the Lemma holds. Then it suffices to consider the case $a \vdash y_1(1)$. Two cases arise:

- $(\neg(a \vdash d))$: by definition of D_3 it follows $(a,d)_{y_1(1)} \in D_3$ and then the Lemma holds.
- $(a \vdash d)$: then $\nexists e \in ya \mid \neg(a \vdash e)$ and then the Lemma holds.

□

Lemma 3. *Let L_0 be in $\mathcal{CFG}$, and let $w_1, w_2 \in L_0$ be sentences representing the same trace $t = [w_1]_{D_2} = [w_2]_{D_2}$, such that for some letters $a \neq b \in \Sigma$, $w_1 = xax_{11}bx_{12}$ and $w_2 = xbx_{21}ax_{22}$, where $x, x_{11}, x_{12}, x_{21}$ and x_{22} are possibly empty strings. Then*

- *$\forall e \in alph(x_{11})$ and $\forall f \in \Sigma$ it is $(b,e)_f \notin D_3(L_0)$;*
- *$\forall e \in alph(x_{21})$ and $\forall f \in \Sigma$ it is $(a,e)_f \notin D_3(L_0)$.*

Comment: consider a program represented by a CFA. Theor. 3 says that it is impossible for a program to have two runs that are a permutation of each other. Moreover this remains impossible for any program that is C-equivalent to the original one.

4 Maximally Parallel Program

In this section we show how to transform a CFA into a semantically equivalent one, and especially into the one with maximal parallelism. For the latter we discuss its relation with the Foata-Cartier normal form of traces.

Ordering by Degree of Parallelism

To compare the degree of parallelism of programs, we introduce two different binary relations over string languages that define the same trace language. We need some definitions. A clique of the independence graph is a set of mutually independent letters. The set of such cliques is denoted by Σ_I. For a string $s \in \Sigma^*$ the *clique decomposition* is $s = s_1 s_2 ... s_n$ where each s_i is a clique, and each s_j, $1 \leq j < n$ contains a letter such that there exists a dependent letter in s_{j+1}. The string decomposition must be computed from left to right, thus ensuring its uniqueness for any given string. The *height* of the string is $height(s) = n$.

Let L_1 and L_2 be string languages. We define an order relation $\geq_P$ on strings $s_1 \in L_1, s_2 \in L_2$ representing the same trace. We say string s_1 is *more parallel* than s_2, written $s_1 \geq_P s_2$, if for their 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 $n \leq m$. This relation induces a partial order on languages: L_1 is more parallel than L_2, written $L_1 \geq_P L_2$, if for each string $s_1 \in L_1$ and $s_2 \in L_2$ with $[s_1]_{D_2(L_1)} = [s_2]_{D_2(L_2)}$, it is $s_1 \geq_P s_2$. The strict relation $>_P$ is defined in the obvious way, using existential quantifier.

Of two strings with equal height, one may place a certain letter in an earlier clique than the other. This concept of greediness or anticipation is captured by another order relation $\geq_G$ on strings $s_1 \in L_1, s_2 \in L_2$ representing the same trace and having equal height. String s_1 is *greedier* than s_2, written $s_1 \geq_G s_2$, if for the clique decompositions $s_1 = s_{1,1}s_{1,2}...s_{1,n}$ and $s_2 = s_{2,1}s_{2,2}...s_{2,n}$ the following condition holds. Let $sign(|s_{1,i}| - |s_{2,i}|)$ be a symbol in $\{0, n, p\}$. Then the sequence of signs is in $0^*p(n \mid p)^*$. The $\geq_G$ naturally induces a partial order on string languages.

We are especially interested in a language that is more parallel and greedy than any other C-equivalent language.

Computing C-Equivalent Languages

We move into application, to present an algorithm for computing the set of C-equivalent automata (vs Def. 5). After proving correctness and completeness of the algorithm, we show how to construct the most parallel and greedy automaton.

Definition 6. *Let A be a CFA with $L(A) \subseteq \Sigma^*$, and let D_3 be the ternary C-dependence relation. The following transformation of the graph of A is termed "moving a letter b to a letter a". Let j be a letter such that:*

$$j \vdash_i b \wedge \forall p \in pred(j) \mid b \dashv p \tag{1}$$

The edges E' of the CFG (Σ, E) of A become as following:

- $E_{del} = \{(p, d) \mid (d = a \vee d = b \vee d = j) \wedge (p, d) \in E\}\}$
- $E_{add} = \{(p, b) \mid (p, a) \in E\} \cup \{(p, j) \mid (p, b) \in E\} \cup \{(p, a) \mid (p, j) \in E\}$
- $E' = (E \setminus E_{del}) \cup E_{add}$

Notice that a becomes the immediate postdominator of b; if $b \in \Sigma_2$, the move can shift other nodes which are transitive successors of b. Such a transformation is called a *legal move* if the new automaton recognizes a language C-equivalent to $L(A)$.

Consider a CFA A and its graph. For each non-initial node a of the graph, we define the *guest set*, $Guest(a) \subseteq \Sigma \setminus \{i, t\}$, which includes the letters b such that:
$\forall c, d \in \Sigma, (a, c)_d \in D_3 \Leftrightarrow (b, c)_d \in D_3$; $\forall c \in \Sigma, (a, c) \in D_{3_2} \Rightarrow (a \dashv c \Leftrightarrow b \dashv c)$ and $\exists j$ such that condition 1 is met. The sets are computed from C-dependence and pre-dominance relations. Since the latter can be computed in polynomial time, the computation of guest sets is in the complexity class P.

The next technical Lemma will be used to prove that we can construct any C-equivalent automaton by moving some letter from its current position in the graph of A, to some node which has the letter as guest.

Lemma 4. *Let L_1 be a $\mathcal{CFG}$ language, the move of b to a is legal if, and only if, $b \in Guest(a)$.*

Theorem 4. *Let $\mathcal{A}$ be the class of automata computed by applying one or more legal moves to A, and $L_0 = L(A)$. Then $\mathcal{A}$ coincides with the class $\mathcal{L}_C(L_0)$.*

The following statement characterizes the pre- and post-dominance properties of guest letters, and will be used to compute the maximally greedy and parallel automaton.

Statement 6. *Let $a, b \in \Sigma$. If $b \in Guest(a)$, i.e. a is a host of b, then the following two relations hold: $b \dashv a \vee a \dashv b$ and $b \vdash a \vee a \vdash b$.*

Maximally Parallel and Greedy Control Flow Automaton

For a given CFA automaton A, we define the *most parallel and greedy* automaton, A_{max}, as the CFA obtained by the following transformation.
For every letter b, legally move b to a letter a such that no other letter c, which pre-dominates a, has b as guest, in formula: $\nexists c \in \Sigma$ such that $b \in Guest(c) \wedge c \dashv a$.

Statement 7. *Let $L = L(A)$ be in $\mathcal{CFG}$ and A_{max} be defined as above. Then $L(A_{max})$ is the most parallel and greedy language C-equivalent to L, i.e. it is: $\forall L' \in \mathcal{L}_C(L)$, $L(A_{max}) >_P L'$ or $L(A_{max}) \geq_G L'$.*

Proof. Since each move applied is legal, i.e. $b \in Guest(a)$, from Theor 4 it follows that the automaton is C-equivalent to A. The proof of maximal greediness is based on Stat. 6.

Foata-Cartier Normal form vs. Maximally Parallel Program

It is known that for a trace $[x]$, the clique decomposition associated to the Foata-Cartier normal form gives the most parallel and greedy form of x. The next example proves that the set of such Foata-Cartier decompositions, in general, is not C-equivalent to the given CFG language. Therefore it differs from $L(A_{max})$, which has been proved to be the most parallel and greedy language which preserves C-equivalence.

For $L \in \mathcal{CFG}$ we denote as A_{Foata} the minimum DFA that recognizes the Foata-Cartier normal forms of the strings in L. More precisely, since there is more than one automaton A_{Foata}, depending on the serialization of the letters in an independence clique, we may assume that serialization complies with the lexicographic order of the terminal alphabet.

Example 1. Let L be the *CFG* language recognized by the CFA A in Figure 2. C-dependence and dominance relations and guest sets are also shown. Let the letters be lexically ordered as listed: $\Sigma = \{i, a, b, c, d, e, t\}$. The Foata-Cartier automaton and A_{max} are in Figure 3.

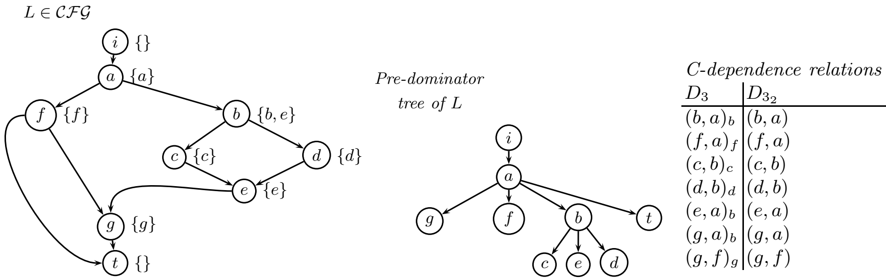


Fig. 2. CFA with relations and guest sets

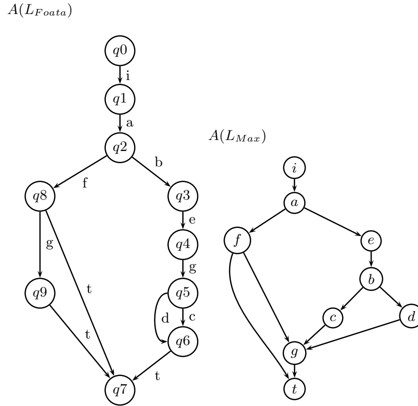


Fig. 3. Foata-Cartier automaton and maximally parallel and greedy automaton

The I-cliques present in the Foata-Cartier strings differ from the cliques in $L(A_{max})$. Clearly A_{Foata} is not C-equivalent to A, since it is not a local automaton. Moreover, this example shows that $L(A_{Foata}) \geq_P L(A_{max})$.

We summarize the above discussions and example in the following statement.

Statement 8. *The automaton A_{Foata} defines the maximally parallel and greedy language $L(A_{Foata})$. The automaton A_{max} defines the maximally parallel and greedy C-equivalent language.*

Technical remark: to achieve the Foata-Cartier parallelism, copies of the same instruction have to be differentiated in order to re-obtain the local property of the automaton. This is an ordinary transformation for compilers, that replicates the same instructions at different addresses.

5 Conclusion

We have shown that program transformations consisting in instruction rescheduling can be conveniently studied and actually implemented using results from

trace theory, combined with concepts and methods of compiler theory. Control-dependence and pre- post-dominance relations play an essential role in that. The automata constructions described are efficient and have a potential for compilation.

Of course data-dependences too must be considered for realistic applications. It is an easy job to superimpose data- onto control constraints. Another straightforward development is to study a program transformation with the aim to achieve uniform parallelism at all steps, instead of maximal greediness.

We believe that this effort for expressing program transformations by means of a suitably enriched trace theory should be continued to cover other parallelizing transformations done by compilers, such as speculative execution, loop unrolling or software pipelining. Such developments are likely to need other algebraic concepts, in particular partially inverse monoids.

Lastly, we observe that we have proved several properties for the traces represented by CFG languages, which are a subclass of local languages. It is not known whether such properties still hold for the latter.

Acknowledgement. we would like to thank Massimiliano Goldwurm for critical reading and suggestions.

References

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D. (eds.): Compilers: principles, techniques, and tools, 2nd edn. Pearson/Addison Wesley, Boston (2007)
2. Appel, A.W., Palsberg, J.: Modern compiler implementation in Java, 2nd edn. Cambridge University Press, Cambridge (2002)
3. Berstel, J., Pin, J.-E.: Local languages and the Berry-Sethi algorithm. Theor. Comput. Sci. 155(2), 439–446 (1996)
4. Bertoni, A., Goldwurm, M., Mauri, G., Sabadini, N.: Counting techniques for inclusion, equivalence and membership problems. In: Diekert, V., Rozenberg, G. (eds.) Counting techniques for inclusion, equivalence and membership problems. The Book of Traces, ch. 5, pp. 131–163. World Scientific, Singapore (1995)
5. Breveglieri, L., Crespi Reghizzi, S., Savelli, A.: Efficient word recognition of certain locally defined trace languages. In: 5th Int. Conf. on Words (WORDS 2005), Montreal, Canada (2005)
6. Diekert, V., Métivier, Y.: Partial commutation and traces. In: Rozenberg, G., Salomaa, A. (eds.) Handbook on Formal Languages, vol. III (1997)
7. Ferrante, J., Ottenstein, K.J., Warren, J.D.: The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9(3), 319–349 (1987)
8. Klarlund, N., Mukund, M., Sohoni, M.: Determinizing asynchronous automata. In: Shamir, E., Abiteboul, S. (eds.) ICALP 1994. LNCS, vol. 820, pp. 130–141. Springer, Heidelberg (1994)
9. Muchnick, S.S.: Advanced compiler design and implementation. Morgan Kaufmann Publishers Inc., San Francisco (1997)
10. Pingali, K., Bilardi, G.: Optimal control dependence computation and the Roman chariots problem. ACM Transactions on Programming Languages and Systems 19(3), 462–491 (1997)