Timing Sensitive Dependency Analysis and its Application to Software Security

Zur Erlangung des akademischen Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN

von der KIT-Fakultät für Informatik des
Karlsruher Instituts für Technologie (KIT)
genehmigte

DISSERTATION

von

Martin Hecker
aus Hannover

Tag der mündlichen Prüfung: 20.07.2020

1. Referent: Prof. Dr.-Ing. Gregor Snelting

2. Referent: Prof. Dr. Markus Müller-Olm

Zusammenfassung

Ich präsentiere neue Verfahren zur statischen Analyse von Ausführungszeit-sensitiver Informationsflusskontrolle in Softwaresystemen. Ich wende diese Verfahren an zur Analyse nebenläufiger Java Programme, sowie zur Analyse von Ausführungszeit-Seitenkanälen in Implementierungen kryptographischer Primitive.

Methoden der Informationsflusskontrolle zielen darauf ab, Fluss von Informationen (z.B.: zwischen verschiedenen externen Schnittstellen einer Software-Komponente) anhand expliziter Richtlinien einzuschränken. Solche Methoden können daher zur Einhaltung sowohl von Vertraulichkeit als auch Integrität eingesetzt werden. Der Ziel *korrekter* statischer Programmanalysen in diesem Umfeld ist der Nachweis, dass in allen Ausführungen eines gegebenen Programms die zugehörigen Richtlinien eingehalten werden. Ein solcher Nachweis erfordert ein Sicherheitskriterium, welches formalisiert, unter welchen Bedingungen dies der Fall ist.

Jedem formalen Sicherheitskriterium entspricht implizit ein Programm- und Angreifermodell. Einfachste *Nichtinterferenz*-Kriterien beschreiben beispielsweise nur nicht-interaktive Programme. Dies sind Programme die nur bei Beginn und Ende der Ausführung Ein- und Ausgaben erlauben. Im zugehörigen Angreifer-Modell kennt der Angreifer das Programm, aber beobachtet nur bestimmte (*öffentliche*) Aus- und Eingaben oder stellt diese bereit. Ein Programm ist nichtinterferent, wenn der Angreifer aus seinen Beobachtungen keinerlei Rückschlüsse auf *geheime* Aus- und Eingaben terminierender Ausführungen machen kann. Aus nicht-terminierenden Ausführungen hingegen sind dem Angreifer in diesem Modell Schlussfolgerungen auf geheime Eingaben erlaubt.

Seitenkanäle entstehen, wenn einem Angreifer aus Beobachtungen *realer* Systeme Rückschlüsse auf vertrauliche Informationen ziehen kann, welche im formalen Modell unmöglich sind. Typische Seitenkanäle (also: in vielen formalen Sicherheitskriterien unmodelliert) sind neben Nichttermination beispielsweise auch Energieverbrauch und die Ausführungszeit von Programmen. Hängt diese von geheimen Eingaben ab, so kann ein Angreifer aus der beobachteten Ausführungszeit auf die Eingabe (z.B.: auf den Wert einzelner geheimer Parameter) schließen.

In meiner Dissertation präsentiere ich neue Abhängigkeitsanalysen, die auch Nichtterminations- und Ausführungszeitkanäle berücksichtigen. In Hinblick auf Nichtterminationskanäle stelle ich neue Verfahren zur Berechnung von Programm-Abhängigkeiten vor. Hierzu entwickle ich ein vereinheitlichendes Rahmenwerk, in welchem sowohl Nichttermination-sensitive als auch Nichttermination-insensitive Abhängigkeiten aus zueinander dualen *Postdominanz*-Begriffen resultieren. Für Ausführungszeitkanäle entwickle ich neue Abhängigkeitsbegriffe und dazugehörige Verfahren zu deren Berechnung. In zwei Anwendungen untermauere ich die These:

Ausführungszeit-sensitive Abhängigkeiten ermöglichen korrekte statische Informationsfluss-Analyse unter Berücksichtigung von Ausführungszeitkanälen.

Basierend auf Ausführungszeit-sensitiven Abhängigkeiten entwerfe ich hierfür neue Analysen für nebenläufige Programme. Ausführungszeit-sensitive Abhängigkeiten sind dort selbst für Ausführungszeit-insensitive Angreifermodelle relevant, da dort *interne* Ausführungszeitkanäle zwischen unterschiedlichen Ausführungsfäden extern beobachtbar sein können. Meine Implementierung für nebenläufige Java Programme basiert auf auf dem Programmanalyse-System JOANA.

Außerdem präsentiere ich neue Analysen für Ausführungszeitkanäle aufgrund *mikro-architektureller* Abhängigkeiten. Exemplarisch untersuche ich Implementierungen von AES256 Blockverschlüsselung. Bei einigen Implementierungen führen Daten-Caches dazu, dass die Ausführungszeit abhängt von Schlüssel und Geheimtext, wodurch diese aus der Ausführungszeit inferierbar sind. Für andere Implementierungen weist meine automatische statische Analyse (unter Annahme einer einfachen konkreten Cache-Mikroarchitektur) die Abwesenheit solcher Kanäle nach.

Abstract

I present new methods for the static analysis of timing sensitive information flow control in software systems. I apply these methods in the analysis of concurrent Java programs, as well as the analysis of timing side-channels in implementations of cryptographic primitives.

Methods for information flow control aim to control the flow of information (e.g.: between different external interfaces of a software component) with respect to explicit flow policies. Such methods can protect confidentiality as well as integrity of information. In this setting, the goal of *sound* static program analysis is to proof that in all executions of a given program, a corresponding flow policy is respected. Such a proof requires a security criterion which formalizes under which condition this is indeed the case.

Every formal security criterion implicitly corresponds to a program and attacker model. Simple *non-interference* criteria, for example, apply only to non-interactive programs. These are programs which allow input and output only at the beginning and the end of the execution. In the corresponding attacker model, the attacker does know the program, but observes only certain (*public*) output and input, or provides those. A program is non-interferent if from these observations the attacker cannot infer any properties of *secret* input and output of terminating executions. From non-terminating executions in this model, the attacker *is* allowed to infer properties of secret input.

Side channels occur if an attacker can infer properties of secret information from observations of *real* systems which are impossible to infer in the formal model. Typical side channels include (i.e., typically unmodeled are): nontermination as well as energy consumption and *timing* channels: By observing the program's execution time, the attacker (partially or completely) infers the program's secret input on which the execution time depends.

In my dissertation I present new dependency analysis sensitive to both nontermination and timing channels. With respect to nontermination, I introduce new methods for the computation of control dependencies. For this purpose, I develop a generalized framework in which nontermination-sensitive as well as nontermination-insensitive control dependencies result from (mutually dual) *postdominance* notions. For timing channels, I develop new notions of control dependencies, and corresponding methods for their computation. In two applications, I substantiate my thesis:

Timing sensitive control dependencies facilitate sound static information flow control in the presence of timing channels.

For this purpose, I develop new analysis for concurrent programs, based on timing sensitive control dependencies. In this setting, timing sensitive control dependencies are relevant even for timing insensitive attacker models, since there *internal* timing channels between concurrent threads of execution may become observable externally. My implementation for concurrent Java programs is based on the JOANA program analysis framework.

I also present new analysis for timing channels caused by *micro-architectural* dependencies. I illustrate these by a study of AES256 block cipher implementations. For some implementations, *data caches* cause the execution time to depend on key and plaintext, making them inferrable from the execution time. For other implementations, my automatic static analysis proves (assuming a simple concrete cache micro-architecture) the absence of such timing channels.

Contents

Symbols

What's all this? Looks like Darth Vader's bathroom.

(Michal Knight — Knight Rider, Knight of the Phoenix)

New contributions of this thesis are **highlighted**.

Part I

nontermination sensitive		p.	p.		**nontermination insensitive**
postdominance	$\sqsupseteq_{\text{MAX}}$	36	36	$\sqsupseteq_{\text{SINK}}$	postdominance
..pseudo-forest	$<_{\text{MAX}}$			$<_{\text{SINK}}$	..pseudo-forest
control dependence	$\rightarrow_{\text{ntscd}}$	34	34	$\rightarrow_{\text{nticd}}$	control dependence
order dependence	$\rightarrow_{\text{ntsod}}$	72	99	$\rightarrow_{\text{ntiod}}$	**order dependence**
"nearest dominator" (generalized)	$\rightarrow_{\text{ntind}}$	122	136	$\rightarrow_{\text{ntsnd}}$	**"nearest dominator" (generalized)**

		p.
standard postdomimance	$\sqsupseteq_{\text{POST}}$	17
standard control dependence	$\rightarrow_{\text{cd}}$	17
decisive order dependence	$\rightarrow_{\text{dod}}$	66
weak order dependence	$\rightarrow_{\text{wod}}$	85
generalized 1-postdominance	$1\text{-}\sqsupseteq$	19
generalized immediate postdominators	$\text{ipdom}_{\sqsupseteq}(n)$	19
generalized postdominance frontier	$\text{PDF}_{\sqsupseteq}(n)$	21
least common ancestor	$\text{lca}_{<}((,n),m)$	46
backward slice	$(\rightarrow_{\text{a}} \mathbin{\dot\cup} \rightarrow_{\text{b}})^*$	74
generic graph transformations	$G_{M\not\rightarrow}, G^{\rightarrow^* M}, \ldots$	75
input equivalence (unlabeled CFG)	$i \sim_S i'$	80
trace equivalence (unlabeled CFG)	$t \sim_S t'$	80
next-observable	$\text{obs}_S(n)$	89
observation equivalance (unlabeled CFG)	$i \sim^{\omega}_{\mathcal{T}_M} i'$	94
weakly deciding nodes	$\text{WD}_G(M)$	139
weak control closure	$\text{WCC}_G(M)$	139

Part II

		p.
timing sensitive postdominance	$\sqsupseteq_{\text{TIME[FIRST]}}$	167
timing sensitive postdominance (transitive)	$\sqsupseteq_{\text{TIME}}$	179
..pseudo-forest	$<_{\text{TIME}}$	179
timing sensitive control dependence	$\to_{\text{tscd}}$	168
"fuel available"	F_n	184
timing dependence	$\to_{\text{td}}$	205

Part III

		p.
timing dependence (in concurrent CFG)	$\to_{\text{timing}}$	331
macro-architectural state	σ_{M}	238
micro-architectural state	σ_{μ}	239
micro-architectural dependence	$\to_{\mu\text{d}}$	254
approximate cache micro-architectural dependence	$\to^{\#}_{\mu\text{d}}$	293

Notation

For any binary relation R, I write R^+ for its transitive closure, and R^* for its transitive reflexive closure. Whenever I denote a binary relation by a symbol similar to $\sqsubseteq$, I write $\sqsupseteq$ for its inverse, and $\sqsubset$ for the relation $\sqsubseteq \setminus \{(x, x)\}$. Similarly for the symbol $\leq$.

```
static int runs = 2000000;
static void main() {
 h = input();
 for (int i=31; i>0; i––) {
  int b = h & (1 << 30);
  A a = new A();
  a.start();
  if (b != 0) {
   delay(runs);
  }
  x = 1;
  h = h << 1;
  a.join();
  print(x);
 }
}

class A extends Thread {
 public void run() {
  delay(runs/2);
  x = 0;
 }
}
```

```
static int delay(int t) {
 int n = 1;
 for (int k=1; k < 100; k++) {
  for (int j=1; j < t; j++) {
   n = n * k;
  }
 }
 return n;
}
```

(a) Internal Timing Channel

```
...
for (int i=0; i<16; i++) {
 state[i] = sbox[state[i]]
}
...
```

(b) External Timing Channel

1 Introduction

Consider the first program code example above. It reliably prints[1] the input value obtained from `h = input()`. It is propagated due to the relative *execution time* of the main thread, and the thread `A` started in the main loop. The execution time of each iteration of the main loop *depends* on the input, and the program exhibits an *internal* timing channel. The second example is part of an implementation of a AES256 block cipher. If run on modern CPUs with *data caches*, and given enough observations of the *execution time* of encrypt operations, an attacker can infer the AES256 key (e.g., [Ber05; BM06]). The implementation exhibits an *external* timing channel.

[1] for example, using OpenJDK11 JVM, on a Intel Xeon Gold 6230 CPU

Timing channel attacks on software systems exploit that the execution time of some program part depends on *secret* input. They occur in numerous attack scenarios. For example:

- The attacker passively observes the execution time of a user-provided program on the users computer. The attacker is either a legitimate client of the program, or monitors the programs network traffic. The attacker may be able to provoke multiple computation with the same secret input, which allows him to take multiple samples of the computation time. This scenario includes several attacks on cryptographic protocols and implementations (e.g., [BB05; Ber05]), such as that shown in the second example.
- The attacker provides a program to the user, who tries to automatically verify that the program leaks no secret information back to the attacker. The attacker may have included code similar to the first example. Alternatively, the user wants to verify that a program provided by himself does not by leak secret information. The verification can be done either by the user himself, or a trusted third party. This scenario includes, for example, every World Wide Web site that includes JavaScript or WebAssembly code, or *app stores* for mobile devices (e.g., [Lor+14]).
- The attackers and the users program run on shared hardware. The attacker is able to observe the users computation through side channels arising from the use of shared resources such as data caches. This scenario includes the “Meltdown” and “Spectre” attacks ([Lip+18; Koc+19]).

In the first two scenarios, timing attacks can in principle be detected by analysis of a single program, while in the third scenario, the interaction of two programs on the shared hardware needs to be taken into consideration. There, the attacker measures the execution time of *his* program.

Information flow control aims to detect and prevent illegitimate flow of information, as defined by a formal security property. Numerous different security properties exist, for example: standard non-interference (for *batch* execution of sequential programs), strategy non-interference (for interactive programs), possibilistic and probabilistic non-interference (for concurrent programs).

Information flow control methods can either be purely static or dynamic. Depending on the program model (abstract syntax tree or control flow graph), static methods are typically based on security type-systems, or program dependency graphs [FOW87; HS09]. Program dependency graphs include some form of *data dependencies* and *control dependencies*. Data dependencies capture the dependencies of the data-state on previous data-state, while control dependencies capture the dependence of the control state on (choices made at) previous control state. Hence data and control dependencies account for *explicit* and *implicit* channels, but not for *timing* channels.

All useful information flow control security properties are proper *hyperproperties*[CS10], which means that their violation cannot be detected purely dynamically (i.e.: by monitoring of one execution of the problem). Therefore, even dynamic information flow control must include *some* form of static analysis. Control dependence can be applied here, as well [Jus+11; XZ07].

In this thesis, I propose a new notion of timing sensitive control dependence. I demonstrate by that this new notion can be used for static timing sensitive information flow control in the first two attack scenarios, and programs like those in the two examples. For the third attack scenario, I suspect that timing sensitive control dependence could in principle be relevant as well, but make no attempt to demonstrate this.

1.1 Contributions

My central contributions (in Part II of this thesis) is the new notion of *timing sensitive control dependence* (Chapter 9), and the related notion of *timing dependence* (Chapter 10). Just as control and data dependencies represent dependencies for data and control state, timing dependencies represent dependencies in the timing state of a configuration, i.e.: the part of an configuration that models how much time has passed during the programs execution. In Chapter 11, I introduce the notion of timing cost model that is implicit in Chapter 9 and Chapter 10. I also introduce the technical notion of *timing stratification*, which sheds light on the relation between timing sensitive control dependence and *nontermination* sensitive control dependence.

In Part III of this thesis I demonstrate that timing dependence and timing sensitive control dependence enable sound static information flow analysis for *internal* timing channels (for concurrent programs, in Chapter 16) and (external) timing channels (due to the micro-architecture of modern CPUs, in Chapter 13). The two central contributions of this part are a new notion of *micro-architectural* dependencies (Section 13.6), and a new timing dependence based criterion for probabilistic noninterference (Section 16.6). I apply *micro-architectural* dependencies in a case study on cache-based timing attacks in implementations of the AES256 block cipher (Chapter 14). At the expense of some precision, I develop an efficient approximation to *micro-architectural* dependencies for cache micro-architectures in Chapter 15. I also explain in Chapter 12 how under simple timing cost models, timing leaks can sometimes be automatically "transformed out".

I derive algorithms for the computation of timing sensitive control dependence and timing dependence from algorithms for *nontermination* sensitive control dependence in arbitrary graphs. My algorithms for micro-architectural dependence on the other hand rely on algorithms for nontermination *insensitive* control dependence and slices. In that light, the central contributions of Part I of thesis then simply are new algorithms (Chapter 3, Chapter 5) for nontermination sensitive and insensitive control dependence (Chapter 4), as well as new algorithms for corresponding slices (Chapter 7). In fact if one is interested only in timing sensitive information flow control, one may want to skip Part I on first reading of this thesis, and trust that indeed: Algorithms for nontermination sensitive and insensitive control dependence in *arbitrary* graphs can be obtained by suitable generalizations of algorithms for standard control dependence in graphs with unique exit node (Section 3.1). But I also want to highlight the following other contributions from Part I, which I find to be of interest independent from their application to timing sensitive analysis: 1. A new fixed-point characterizations of nontermination sensitive and insensitive postdominance (Section 5.1). 2. Two new notions for nontermination sensitive and insensitive *order* dependence (Chapter 6). 3. Two new notions of soundness for nontermination insensitive slicing (Section 6.5, Section 6.6). 4. Reductions of several different notions of nontermination insensitive slicing to slices from nontermination insensitive control dependence (Section 7.5).

2 Methodology

The first principle is that you must not fool yourself — and you are the easiest person to fool. So you have to be very careful about that. And after you've not fooled yourself, it's easy not to fool other scientists. You just have to be honest in a conventional way after that.

(Richard P. Feynman — Cargo Cult Science)

In this thesis, I claim numerous new results (e.g., soundness and sometimes minimality) for various forms of program dependencies. The conventional method to support such claims is by mathematical proof, which can range from offhand sketches to fully formalize and mechanically checked proof scripts for systems like Isabelle/HOL ([NWP02]), Coq ([Tea17]) or Lean ([Mou+15]). In this thesis, I provide semi-formal proofs for some, but by no means for all new results. For most other results, I gathered evidence by performing extensive *random testing* of the claimed result. Specifically, for such results I

- implemented all involved predicates and algorithms in the pure functional programming language Haskell,
- used a randomized test data generator to generate *inputs*, and
- verified the claimed result for a large number of random inputs.

For example, Observation 5.3.7 involves one algorithm and a relation $\sqsupseteq_{\text{SINK}}$, and reads:

Let G be any CFG. Then Algorithm 6 terminates with a result $<_{\text{SINK}}$ s.t. $>_{\text{SINK}}$ is a transitive reduction of $\sqsupseteq_{\text{SINK}}$.

In order to randomly test this observation, I use a rather inefficient "reference" implementation of the relation $\sqsupseteq_{\text{SINK}}$ to check the more efficient to Algorithm 6. The "input" graphs G are obtained from the randomized graph generator for the Haskell Functional Graph Library ([EM17; Mil17; Erw01]). I then test the observation on random graphs of size up to 100 nodes, by running the random tests for several days. Here, one million of such graphs can tested in approximately one hour on a standard desktop PC.

In this example I cannot use a naive "reference" implementation of the relation $\sqsupseteq_{\text{SINK}}$. In fact, a direct implementation of its definition would require me to check infinitely many paths in input graphs G. Instead I use a fixed point characterization of $\sqsupseteq_{\text{SINK}}$ (Theorem 5.1.2) which *can* be naively implemented in Haskell. While this reference implementation is effective, it is not very efficient for larger graphs (e.g.: larger than 100 nodes). For example, it would take me at least 16 hours to check this observation for one million random graphs of size up to 500 nodes.

For every **Observation** in this thesis, I provide random test properties in the Haskell QuickCheck framework ([CH02]). For example, the property for Observation 5.3.7 reads[1]

```
observation_5_3_7 = testProperty "isinkdomOf^* == sinkdomOfGfp" $
  \(ARBITRARY(g)) ->
       toSuccMap $ trc $ (fromSuccMap $ PDOM.isinkdomOf g)
    == PDOM.sinkdomOfGfp g
```

All properties are available in a ready-to-run virtual machine image[Hec20]. They can be found in the Haskell module `Program.Properties.DissObservations`.

[1] up to renaming of variables, and type annotations

Ideally, of course, every observation would be supported not only by evidence from randomized tests, but by formal proofs. In fact for some key results of Part I, proofs in the Isabelle/HOL proof assistant are available due to ongoing work by Simon Bischof[Bis19].

Part I

Dependency Analysis in Arbitrary Graphs

Choice. The problem is: choice.

(Neo, The Matrix Reloaded (2003), The Wachowskis)

3 Control Dependence in Arbitrary Graphs

I am never forget the day I first meet the great Lobachevsky. In *one* word he told me secret of success in mathematics: Plagiarize!

Let no one else's work evade your eyes
Remember why the good Lord made your eyes
So don't shade your eyes
But plagiarize, plagiarize, plagiarize

Only be sure always to call it please "research"

(Tom Lehrer — Lobachevsky)

My overall strategy to obtain definitions and algorithms for time sensitive control dependence is as follows:

1. Generalize existing algorithms for standard (nontermination insensitive) control dependence such that they become applicable also to nontermination sensitive control dependence.
2. Define timing sensitive control dependence as modification of nontermination sensitive control dependence. Then modify the generalized algorithms for nontermination sensitive control dependence such that they become applicable also to timing sensitive control dependence.

In principle, I could have tried to follow this strategy under the customary assumption that (control flow) graphs always come with a *unique* exit node. Instead I chose to drop this assumption, and develop generalizations that apply to *arbitrary* graphs. I do this not only out of theoretical curiosity, since this investment pays off at the latest in Chapter 13, in which I will utilize control dependence for certain graphs (*derived* from control flow graphs) that do *not* have unique exit nodes, even if the control flow graphs they derive from do have a unique exit node.

In Section 3.1 of this chapter, I define the framework of generalized control dependence for arbitrary graphs, of which standard control dependence is an instantiation. Then in Section 3.2, I develop a new algorithm for the computation of generalized control dependence. I do so by generalizing the standard approach from [Cyt+91].

3.1 Generalized Control Dependence

For the purpose of the first two parts of this thesis, a *control flow graph* is any directed, unlabeled graph. Standard definitions and algorithm for control dependence require the existence of *unique* exit-nodes, but the new generalized definitions introduced in this chapter do not.

Definition 3.1.1 (Control Flow Graph)**.** A pair $G = (N, E)$ of a finite set N of nodes, and a set $E \subseteq N \times N$ of directed edges is called a *control flow graph*. I write

- $n \to_G m$ whenever $(n, m) \in E$, and sometimes omit the subscript G when it is clear from the context: $n \to m$.

A node $n_e \in N$ is called the *unique entry node* of G whenever it is the only in N such that

- it is an entry node, i.e.: n_e has no incoming edge in G, and
- every node is reachable from n_e, i.e.: $\forall m \in N.\ n_e \to^* m$

Similarly, a node $n_x \in N$ is called the *unique exit node* of G whenever it is the only in N such that

- it is an exit node, i.e.: n_x has no outgoing edge in G, and
- n_x is reachable from every node, i.e.: $\forall m \in N.\ m \to^* n_x$

A path $\pi = n_1, \ldots, n_k, \ldots$ is a (possibly empty or infinite) sequence of nodes $n_i \in N$ such that $(n_i, n_{i+1}) \in E$ for all $n_i, n_{i+1} \in \pi$. Given a finite path $\pi_1 = n_1, \ldots, n_k$ and a path $\pi_2 = m_1 \ldots,$ I write π_1, π_2 for

- the path $n_1, \ldots, n_k, m_1, \ldots$ if neither π_1 nor π_2 is empty, and $(n_k, m_1) \in E$
- the path π_2 if π_1 is empty
- the path π_1 if π_2 is empty

A finite path $\pi = n_0, \dots, n_k$ is said to have length k. A finite path π of length k is said to be between n and m if $n = n_0$ and $m = n_k$. In this case, I write $n \overset{\pi}{\rightarrow}^{k} m$, and sometimes omit k. With ${}_{n}\Pi_{m}$ I denote the set of all (finite) paths between n and m.

Intuitively, a node n (immediately) controls whether a node m is executed, if a (control flow) *choice* made at n — i.e.: the choice at which successor n_l of some choice node n to continue — can force control flow to "eventually" reach m, while a different choice (to continue at some other successor n_r of n) does *not* force control flow to eventually reach m.

This intuitive characterization is ambiguous: what exactly does it mean for control flow at n_l to be forced to "eventually" reach m? Different realization of this concept will lead to different control dependence notion. Formally, control dependence is parameterized by a binary relation $\sqsupseteq$ on nodes that specifies a notion of *eventuality*.

Definition 3.1.2 ($\sqsupseteq$-Control Dependence)**.** Let $G = (N, E)$ be a CFG, and $\sqsupseteq \subseteq N \times N$ any binary relation on nodes[1] (where $m \sqsupseteq n$ is meant to be read: any path from n "eventually" reaches m, or: m "postdominates" n). Then m is said to be $\sqsupseteq$-control dependent on n in G iff there exist nodes n_l and n_r such that

- $n \rightarrow n_l$ and $n \rightarrow n_r$,
- $m \sqsupseteq n_l$, and
- $\neg\, m \sqsupseteq n_r$

One contribution of this thesis is a new *timing sensitive* instantiation of $\sqsupseteq$, which I will introduce in Section 9.1. For now, I will review the instantiation of $\sqsupseteq$ that leads to *standard* control dependence.

Given a graph G with unique exit node n_x, it is natural to say that *any* path "eventually" reaches n_x. For arbitrary nodes m, it is then equally

[1] more formally: a function that maps a CFG G to such a relation $\sqsupseteq$

natural to say that any path from n reaches m if m *postdominates* n (w.r.t the unique exit node n_x), i.e.: if m occurs on every (necessarily finite) path between n and n_x:

Definition 3.1.3 (Standard Postdominance)**.** Let G be a CFG with unique exit node n_x. Then m (standard) postdominates n, and I write $m \sqsupseteq_{\text{POST}} n$, iff all paths in G from n to n_x contain m, i.e.: iff

$$\forall \pi \in {}_n\Pi_{n_x}.\ m \in \pi$$

Also, for *standard* control dependence (i.e.: $\sqsupseteq_{\text{POST}}$-control dependence) I just write:

$$n \to_{\text{cd}} \ m$$

Standard control dependence $n \to_{\text{cd}} \ m$ then holds iff there exist successors n_l and n_r of n such that

- $m \sqsupseteq_{\text{POST}} n_l$ (i.e.: m standard postdominates n), and
- $\neg\ m \sqsupseteq_{\text{POST}} n_r$

Remark 3.1.1. This definition of standard control dependence $n \to_{\text{cd}}$ m appears to be due to [Wol95]. A more classical definition is found in [FOW87], and reads:

Definition 3.1.4 (Standard Control Dependence, Alternative Definition)**.** m is standard control dependent on n, iff

1. $\neg\ m \sqsupseteq_{\text{POST}} n$
2. there exists a path $n \overset{n,\pi,m}{\to} \ m$ such that
   - $m \notin \pi$, and
   - $\forall n' \in \pi.\ m \sqsupseteq_{\text{POST}} n'$

A proof of equivalence of these two definition can be found in [Was10].

3.2 An Algorithm for Generalized Control Dependence

Efficient computation of standard control dependence for CFGs with unique exit node n_x is possible due to the following facts:

1. Every node $n \neq n_x$ has a unique *immediate* post-dominator $\text{ipdom}\,(n) \in N$, i.e. a node m such that $m \sqsupset_{\text{POST}} n$, and $m' \sqsupseteq_{\text{POST}} m$ for all nodes m' s.t. $m' \sqsupset_{\text{POST}} n$.
2. Hence, the mapping ipdom induces a tree rooted in n_x.
3. This *postdominator tree* can be computed efficiently (e.g., [LT79]).
4. Given the postdominator tree, the *postdominance frontier* $\text{PDF}\,(m) \subseteq N$ for nodes $m \in N$ can be computed efficiently ([Cyt+91]).

   Here, $\text{PDF}\,(m)$ is the set of all nodes n such that

   - m does not strictly post-dominate n
   - m post-dominates some direct successor n' of n,
5. $n \rightarrow_{\text{cd}} m$ iff $n \in \text{PDF}\,(m)$ for $m \neq n$.

In order to understand item 5, observe that item 2 of Definition 3.1.4 is equivalent to m postdominating some direct G-successor n' of n ([Cyt+91], Lemma 11).

In this section, I develop suitable generalizations of the notions *post-dominance* and *postdominance frontier*, and present a new generalized algorithm for the *computation* of postdominance frontiers. This algorithm makes use of a generalization of postdominator trees. Later in Section 4.1, I will provide efficient algorithms for their computation in *arbitrary* CFGs (specifically: CFGs which lack a unique exit node n_x), and hence obtain an efficient algorithm for the computation of control dependence in arbitrary CFGs.

Generalized Immediate Postdominance

The algorithm from [Cyt+91] computes the postdominance frontiers for the standard postdominance relation $\sqsupseteq_{\text{POST}}$[2]. In this subsection, I identify a set of sufficient properties of otherwise arbitrary postdominance relations $\sqsupseteq$ that are enough to allow efficient postdominance frontiers algorithms for such arbitrary postdominace relations. Specifically, these postdominance relations will not need to be anti-symmetric (as $\sqsupseteq_{\text{POST}}$ is). In a first step, I provide a substitute for the notion of *strict* postdominance.

Definition 3.2.1 (Immediate $\sqsupseteq$-Postdominance)**.** Given a relation $\sqsupseteq \subseteq N \times N$, a node $x \in N$ is said to 1-$\sqsupseteq$-postdominate z if there exists some node y such that

$$x \sqsupset y \sqsupseteq z$$

The set $\text{ipdom}_{\sqsupseteq}(n)$ of *immediate* $\sqsupseteq$*-postdominators* of n is defined by

$$\text{ipdom}_{\sqsupseteq}(n) = \left\{ m \;\middle|\; \begin{array}{c} m \;\; 1\text{-}{\sqsupseteq}\; n \\ \forall m' \in N.\, m' \; 1\text{-}{\sqsupseteq}\; n \quad \Longrightarrow \quad m' \sqsupseteq m \end{array} \right\}$$

Remark 3.2.1. 1-$\sqsupseteq$ differs from strict postdominance $\sqsupset$ whenever $\sqsupseteq$ contains “cycles”. If, for example,

$$x \sqsupset y \sqsupset x$$

then x 1-$\sqsupseteq$ x, but not: $x \sqsupset x$. If, on the other hand, $\sqsupseteq$ is a partial order, then

$$m \sqsupset n \quad \Leftrightarrow m \; 1\text{-}{\sqsupseteq}\; n$$

Corollary 3.2.1. Immediate $\sqsupseteq$-postdominance generalizes immediate postdominance for CFGs with unique exit node n_x:

- $\text{ipdom}_{\sqsupseteq_{\text{POST}}}(n_x) = \emptyset$

[2] or the dominance frontiers for the fixed dominance relation $\sqsupseteq_{\text{DOM}}$, which is exactly the postdominance relation in the graph G^{-1} with inverted edges

- $\mathrm{ipdom}_{\sqsupseteq_{\mathrm{POST}}}(n) = \{\mathrm{ipdom}\,(n)\}$ for $n \neq n_x$

Remark 3.2.2. Essentially, 1-$\sqsupseteq$-postdominance behaves like standard strict postdominance, but allows sets of "equivalent" (w.r.t $\sqsupseteq$) nodes, i.e. nodes m, m' such that $m \sqsupset m'$ and $m' \sqsupset m$.

Informally:

$$\begin{array}{lcl} 1\text{-}\sqsupseteq & \overset{\wedge}{=} & \sqsupset_{\mathrm{POST}} \text{ for relations possibly lacking anti-symmetry} \\ \mathrm{ipdom}_{\sqsupseteq} & \overset{\wedge}{=} & \mathrm{ipdom}_{\sqsupseteq_{\mathrm{POST}}} \text{ lifted to sets of } \sqsupseteq \text{-equivalent nodes} \end{array}$$

Specifically, if $\sqsupseteq$ is reflexive and transitive, it admits the following rules:

Lemma 3.2.1.

$$\frac{x \sqsupseteq y \qquad y \sqsupseteq x \qquad x \neq y}{x \in \mathrm{ipdom}_{\sqsupseteq}(y) \ \wedge y \in \mathrm{ipdom}_{\sqsupseteq}(x)}\ \mathrm{EQ}_1^{\sqsupseteq}$$

$$\frac{x \sqsupseteq y \qquad y \sqsupseteq x}{\mathrm{ipdom}_{\sqsupseteq}(x) = \mathrm{ipdom}_{\sqsupseteq}(y)}\ \mathrm{EQ}_2^{\sqsupseteq}$$

$$\frac{x \sqsupseteq y \qquad y \sqsupseteq x \qquad x \in \mathrm{ipdom}_{\sqsupseteq}(z)}{y \in \mathrm{ipdom}_{\sqsupseteq}(z)}\ \mathrm{EQ}_3^{\sqsupseteq}$$

Rule $\mathrm{EQ}_1^{\sqsupseteq}$ says that if x and y are in a $\sqsupseteq$-"cycle", then they are immediate $\sqsupseteq$-postdominators of each other. By $\mathrm{EQ}_2^{\sqsupseteq}$, they then also have the same set of immediate $\sqsupseteq$-postdominators. $\mathrm{EQ}_3^{\sqsupseteq}$ says that if x is an immediate $\sqsupseteq$-postdominator of some other node y, then also is y.

Proof: On page 350 in the appendix. □

Postdominance Frontiers

Intuitively, the postdominance frontier of a node x is the set of nodes y such that y is at the frontier of regions of nodes postdominating x, i.e.: only *after* making a CFG choice $y \rightarrow_G s$ at y am I guaranteed to "eventually" meet x. I propose the following general definition:

Definition 3.2.2 ($\sqsupseteq$-Postominance Frontiers)**.** Given a CFG $G = (N, E)$, a relation $\sqsupseteq \subseteq N \times N$ and a node $x \in N$, the $\sqsupseteq$-postdominance frontier $\mathrm{PDF}_{\sqsupseteq}(x)$ is defined by

$$\mathrm{PDF}_{\sqsupseteq}(x) = \left\{ y \;\middle|\; \begin{array}{r} \neg\, x \text{ 1-}\sqsupseteq y \\ \text{for some } s \text{ s.t. } y \rightarrow_G s : \quad x \sqsupseteq s \end{array} \right\}$$

The key to efficient computation of $\mathrm{PDF}_{\sqsupseteq}$ is to partition $\mathrm{PDF}_{\sqsupseteq}(x)$ into two parts: those nodes y contributed to $\mathrm{PDF}_{\sqsupseteq}(x)$ *locally*, and those nodes y contributed to $\mathrm{PDF}_{\sqsupseteq}(x)$ by nodes z which are immediately $\sqsupseteq$-postdominated by x (implying $x \sqsupseteq z$).

Definition 3.2.3 ($\sqsupseteq$-Postdominance Frontiers: *local* part)**.** Given a CFG $G = (N, E)$, a relation $\sqsupseteq \subseteq N \times N$ and a node $x \in N$, the $\sqsupseteq$-postdominance frontiers *local* part $\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(x)$ is defined by

$$\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(x) = \left\{ y \;\middle|\; \begin{array}{r} \neg\, x \text{ 1-}\sqsupseteq y \\ y \rightarrow_G x \end{array} \right\}$$

Definition 3.2.4 ($\sqsupseteq$-Postdominance Frontiers: *up* part)**.** Given a CFG $G = (N, E)$, a relation $\sqsupseteq \subseteq N \times N$ and a node $z \in N$, the part $\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z)$ of nodes $y \in \mathrm{PDF}_{\sqsupseteq}(z)$ that z contributes *upwards* (i.e.: to immediate $\sqsupseteq$-postdominators x of z) is defined by

$$\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z) = \left\{ y \in \mathrm{PDF}_{\sqsupseteq}(z) \;\middle|\; \forall x \in \mathrm{ipdom}_{\sqsupseteq}(z) \,.\, \neg\, x \text{ 1-}\sqsupseteq y \right\}$$

Remark 3.2.3. Definition 3.2.2, 3.2.3 and 3.2.4 generalize definitions from ([Cyt+91], Section 4.2), by parameterizing w.r.t the underlying postdominance relation $\sqsupseteq$, and replacing $\sqsupset$ by the generalized notion of 1-$\sqsupseteq$ -postdominance.

Specifically, I have:

$$\mathrm{PDF}_{\sqsupseteq_{\mathrm{POST}}}(x) = \left\{ y \,\middle|\, \begin{array}{rr} & \neg\, x \sqsupset_{\mathrm{POST}} y \\ \text{for some } s \text{ s.t. } y \to_G s : & x \sqsupseteq_{\mathrm{POST}} s \end{array} \right\}$$

$$\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq_{\mathrm{POST}}}(x) = \left\{ y \,\middle|\, \begin{array}{r} \neg\, x \sqsupset_{\mathrm{POST}} y \\ y \to_G x \end{array} \right\}$$

$$\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq_{\mathrm{POST}}}(z) = \left\{ y \in \mathrm{PDF}_{\sqsupseteq_{\mathrm{POST}}}(z) \,\middle|\, \neg\ \mathrm{ipdom}(z) \sqsupset_{\mathrm{POST}} y \right\}$$

Note that in [Cyt+91], dominance frontiers are defined for *pre*dominance, and CFGs with unique entry node n_e. In contrast, I only consider (generalizations of) *post*dominance. Consequently, my definitions differ from those in [Cyt+91] also w.r.t the direction of CFG edges.

I now state the conditions under which $\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}$ and $\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}$ are indeed suitable partitions of $\mathrm{PDF}_{\sqsupseteq}$.

Lemma 3.2.2. Assume a CFG $G = (N, E)$, a node $z \in N$ and a relation $\sqsupseteq \ \subseteq \ N \times N$ such that $\sqsupseteq$ is transitive and reflexive. Also, identify $\mathrm{ipdom}_{\sqsupseteq}$ with the relation $\left\{(x,z) \,\middle|\, x \in \mathrm{ipdom}_{\sqsupseteq}(z)\right\}$, and assume that its transitive and reflexive closure is $\sqsupseteq$, i.e.:

$$\mathrm{ipdom}^{*}_{\sqsupseteq} = \sqsupseteq$$

Then:

$$\mathrm{PDF}_{\sqsupseteq}(x) = \mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(x) \quad \cup \bigcup_{\{z \,\mid\, x \in \mathrm{ipdom}_{\sqsupseteq}(z)\}} \mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z)$$

Proof: On page 350 in the appendix. □

Remark 3.2.4. For CFG with unique exit node n_x, the requirement

$$\mathrm{ipdom}^*_{\sqsupseteq} = \sqsupseteq$$

for $\sqsupseteq = \sqsupseteq_{\mathrm{POST}}$ is trivially true because there, the unique transitive reduction $>$ of $\sqsupseteq_{\mathrm{POST}}$ (i.e.: the $\sqsupseteq_{\mathrm{POST}}$-dominance tree) is exactly $\mathrm{ipdom}_{\sqsupseteq_{\mathrm{POST}}}$:

$$m > n \quad \Leftrightarrow \quad m \in \mathrm{ipdom}\,(n)$$

This equivalence between $\mathrm{ipdom}_{\sqsupseteq}$ and transitive reductions of $\sqsupseteq$ will *not* hold in general.

Algorithmically, the definitions of $\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}$ and $\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}$ are not satisfactory, for they imply reachability checks

$$x \; 1\text{-}\sqsupseteq \; y$$

Now I identify the conditions under which these checks can by replaced by cheaper checks that only involve *immediate* $\sqsupseteq$-postdominators.

Definition 3.2.5 (Closed under $\rightarrow_G$). A relation $\sqsupseteq \;\subseteq N \times N$ is said to be closed under $\rightarrow_G$ if it admits the rule:

$$\frac{y \rightarrow_G x \qquad x' \sqsupseteq y \qquad x' \neq y}{x' \sqsupseteq x}\,\mathrm{CL}^{\rightarrow_G}$$

Remark 3.2.5. Of course, $\sqsupseteq_{\mathrm{POST}}$ is closed under $\rightarrow_G$.

Lemma 3.2.3. Let $\sqsupseteq \;\subseteq N \times N$ be transitive, and closed under $\rightarrow_G$. Then

$$\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(x) = \left\{ y \;\middle|\; \begin{array}{r} \neg\, x \in \mathrm{ipdom}_{\sqsupseteq}(y) \\ y \rightarrow_G x \end{array} \right\}$$

Proof: On page 352 in the appendix. □

Informally, the next technical observation says that (unless x and y are $\sqsupseteq$-equivalent) when starting at y and going $\sqsupseteq$-upwards towards x, there exist some nodes w, v such that I am not $\sqsupseteq$-equivalent to x upon reaching v, but *become* $\sqsupseteq$-equivalent to x by taking just one more $\geq$ step towards x (from v to w).

Observation 3.2.1. Let $\geq$ be any relation such that $\geq^* = \sqsupseteq$, and $x \sqsupseteq y$ but $\neg y \sqsupseteq x$. Then there exists some w, v such that

$$x \sqsupseteq w \geq v \sqsupseteq y$$

$$\begin{array}{lccc} \text{and} & w & \sqsupseteq & x \\ \text{but: } \neg & v & \sqsupseteq & x \end{array}$$

The requirement of $\sqsupseteq$ being closed under $\to_G$ allowed me to simplify $\text{PDF}^{\text{local}}_{\sqsupseteq}$. The next additional requirement will allow me to simplify $\text{PDF}^{\text{up}}_{\sqsupseteq}$.

Definition 3.2.6 (Lack of Joins)**.** A relation $\sqsupseteq \;\subseteq N \times N$ is said to *lack joins*, if it admits the rule:

$$\frac{\begin{array}{ccc} x \in \text{ipdom}_{\sqsupseteq}(v) & v \sqsupseteq s & \\ x \in \text{ipdom}_{\sqsupseteq}(z) & z \sqsupseteq s & z \neq v \end{array}}{v \in \text{ipdom}_{\sqsupseteq}(z) \;\vee z \in \text{ipdom}_{\sqsupseteq}(v)}\;\text{NoJoin}$$

In order to justify the phrase *lacking joins*, consider Figure 3.1, where arrows $n \to m$ signify[3] $m \in \text{ipdom}_{\sqsupseteq}(n)$. Rule NoJoin demands that in Figure 3.1a, there are no joins at x for "paths" starting in s, unless

[3] i.e.: arrows are *downwards* with respect to $\sqsupseteq$, that is, they are in accord with the inverse $\sqsubseteq$ of $\sqsupseteq$. The justification of this choice of presentation is that it presents postdominance relations (or their transitive reductions) in accord with the "general direction" of control flow. Specifically, if in a CFG I have an edge $n \to_G m$ such that m is the *only* G-successor of n, I will have $m \in \text{ipdom}_{\sqsupseteq}(n)$ (and usually: $m > n$ in a transitive reduction of $\sqsupseteq$). By drawing an edge from n to m, this then is in accord with the G-edge $n \to_G m$.

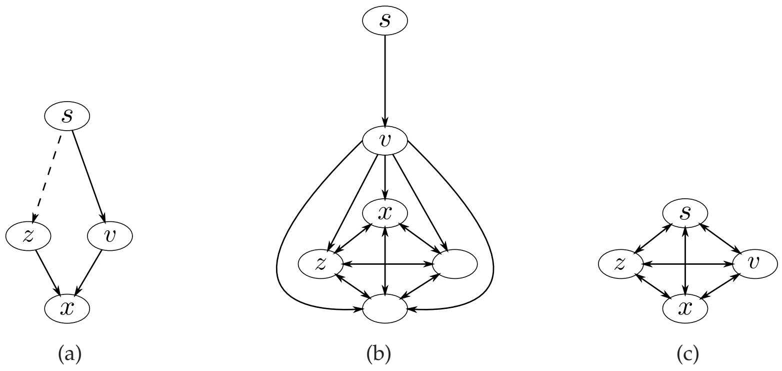


Figure 3.1: Illustration of Definition 3.2.6.

either $z \to v$ or $v \to z$, i.e.: I cannot have $s \to z$ in Figure 3.1a. On the other hand, the situations in Figure 3.1b and Figure 3.1c are allowed.

Remark 3.2.6. Standard postdominance $\sqsupseteq_{\text{POST}}$ lacks joins, because (given a unique exit node n_x) $\text{ipdom}_{\sqsupseteq_{\text{POST}}}$ forms a tree rooted in n_x, and $\sqsupseteq_{\text{POST}} = \text{ipdom}^*$.

Lemma 3.2.4. Let $\sqsupseteq \ \subseteq N \times N$ be transitive, reflexive, lacking joins, and closed under $\to_G$. Also assume

$$\text{ipdom}^*_{\sqsupseteq} = \sqsupseteq$$

Then, given some $x \in \text{ipdom}_{\sqsupseteq}(z)$:

$$\text{PDF}^{\text{up}}_{\sqsupseteq}(z) = \left\{ y \in \text{PDF}_{\sqsupseteq}(z) \;\middle|\; \neg\, x \in \text{ipdom}_{\sqsupseteq}(y) \right\}$$

Proof: On page 352 in the appendix. □

Definition 3.2.7 (Efficient PDF Partitioning)**.** In summary, I say that a relation $\sqsupseteq$ *admits an efficient* PDF *partitioning* if

1. it is reflexive and transitive (but not necessarily anti-symmetric)

2. $\text{ipdom}^*_{\sqsupseteq} = \sqsupseteq$
3. it admits the rules

$$\frac{y \rightarrow_G x \qquad x' \sqsupseteq y \qquad x' \neq y}{x' \sqsupseteq x}\,\text{CL}^{\rightarrow_G}$$

$$\frac{\begin{array}{cc} x \in \text{ipdom}_{\sqsupseteq}(v) & v \sqsupseteq s \\ x \in \text{ipdom}_{\sqsupseteq}(z) & z \sqsupseteq s \end{array} \quad z \neq v}{v \in \text{ipdom}_{\sqsupseteq}(z) \;\vee z \in \text{ipdom}_{\sqsupseteq}(v)}\,\text{NoJoin}$$

In that case, I have

$$\text{PDF}_{\sqsupseteq}(x) = \text{PDF}^{\text{local}}_{\sqsupseteq}(x) \quad \cup \quad \bigcup_{\{z \mid x \in \text{ipdom}_{\sqsupseteq}(z)\}} \text{PDF}^{\text{up}}_{\sqsupseteq}(z)$$

$$\text{PDF}^{\text{local}}_{\sqsupseteq}(x) = \left\{ y \;\middle|\; \begin{array}{r} \neg\, x \in \text{ipdom}_{\sqsupseteq}(y) \\ y \rightarrow_G x \end{array} \right\}$$

$$\text{PDF}^{\text{up}}_{\sqsupseteq}(z) = \left\{ y \in \text{PDF}_{\sqsupseteq}(z) \;\middle|\; \neg\, x \in \text{ipdom}_{\sqsupseteq}(y) \right\} \quad \text{for } x \in \text{ipdom}_{\sqsupseteq}(z)$$

If $\sqsupseteq$ admits an efficient PDF partitioning, an algorithm to compute PDF is immediately available by

1. computing $\text{ipdom}_{\sqsupseteq}$
2. computing — using, e.g., any suitable work-list algorithm — the least fixed point of the monotone functional defined by the rules

$$\frac{\neg\, x \in \text{ipdom}_{\sqsupseteq}(y) \qquad y \rightarrow_G x}{y \in \text{PDF}_{\sqsupseteq}(x)}$$

$$\frac{\neg\, x \in \text{ipdom}_{\sqsupseteq}(y) \qquad x \in \text{ipdom}_{\sqsupseteq}(z) \qquad y \in \text{PDF}_{\sqsupseteq}(z)}{y \in \text{PDF}_{\sqsupseteq}(x)}$$

It is natural to choose a *topological* iteration order during fixed-point iteration, and also to choose an efficient representation of $\text{ipdom}_{\sqsupseteq}$. A naive representation of $\text{ipdom}_{\sqsupseteq}$ (e.g.: using a generic data structure that maps nodes to set of nodes) is inefficient in general: recall from rule $\text{EQ}_3^{\sqsupseteq}$ and $\text{EQ}_2^{\sqsupseteq}$ that $\text{ipdom}_{\sqsupseteq}(z)$ may consists of many $x \neq x'$ such that $x \sqsupseteq x'$ and $x' \sqsupseteq x$, and also that then: $\text{ipdom}_{\sqsupseteq}(x) = \text{ipdom}_{\sqsupseteq}(x')$.

Given *any* transitive reduction $>$ of $\sqsupseteq$, I can

1. compute the strongly connected components of the graph $(N, <)$, and a corresponding topological sorting, both of which can either be provided implicitly by the algorithm computing $<$, or can be done simultaneously by, e.g., Tarjan's algorithm [Tar72].
2. compute PDF by traversing the condensed graph in that order *once*.

For the computation of *nontermination insensitive* control dependence and *nontermination sensitive* control dependence (Section 4.1), I will use algorithm 1. Note that the test

$$\neg\ x \in \text{ipdom}_{\sqsupseteq}(y)$$

becomes

$$\neg \exists x' \in \text{scc}.\ x' > y$$

due to the characterizations of $\text{ipdom}_{\sqsupseteq_{\text{SINK}}}$ and $\text{ipdom}_{\sqsupseteq_{\text{MAX}}}$ in Observation 5.3.2 and Observation 5.2.2. Also note that by computing the set $\text{scc}_{>} = \{\, y \mid \exists x' \in \text{scc}.\ x' > y \,\}$ *once* per scc, I can use this for both the tests on y, and for enumerating z.

Of course, for $\sqsupseteq\ = \sqsupseteq_{\text{POST}}$, algorithm 1 becomes the well known algorithm 2 from [Cyt+91].

```
Input  : A transitive reduction $<$ of $\sqsubseteq$
Input  : A topological sorting sccs of all strongly connected
         components of $<$.
Output: PDF$_\sqsupseteq$
for scc $\in$ sccs do
    local $\leftarrow \{y \mid x \in \mathsf{scc},\ y \rightarrow_G x,\ \neg \underbrace{\exists x' \in \mathsf{scc}.\ x' > y}_{y \in \mathsf{scc}_>}\}$
    up    $\leftarrow \{y \mid \underbrace{x \in \mathsf{scc},\ x > z}_{z \in \mathsf{scc}_>},\ y \in \mathsf{DF}[z],\ \neg \underbrace{\exists x' \in \mathsf{scc}.\ x' > y}_{y \in \mathsf{scc}_>}\}$
    for $x \in$ scc do
        $\mathsf{DF}[x] \leftarrow$ local $\cup$ up
    end
end
```

Algorithm 1: Computation of PDF

```
Input  : The postdominance tree $>$
Output: PDF$_{\sqsupseteq_{\text{POST}}}$
for each node $x$ in a bottom up traversal of $>$ do
    local $\leftarrow \{y \mid y \rightarrow_G x,\ \neg\ x > y\}$
    up    $\leftarrow \{y \mid x > z,\ y \in \mathsf{DF}[z],\ \neg\ x > y\}$
    $\mathsf{DF}[x] \leftarrow$ local $\cup$ up
end
```

Algorithm 2: Computation of PDF$_{\sqsupseteq_{\text{POST}}}$, [Cyt+91]

3.3 Related Work

I generalized from standard postdominance $\sqsupseteq_{\mathrm{POST}}$ to relations $\sqsupseteq$ that admit an efficient PDF partitioning. Specifically, they need not be anti-symmetric, and CFGs are not required to have a unique exit node n_x. Other generalization exists, but still require a unique exit node n_x. For example, the authors of [BP96] generalize to P-dominance relations, and identify conditions that guarantee that the dominance relation remains anti-symmetric, and that its transitive reduction is unique and forest-structured. They apply their framework to obtain algorithms to compute "weak" control dependence. The framework from [CR06] also only applies to CFG with unique exit node n_x.

Summary

- The standard algorithm by Cytron et al for standard control dependence can be generalized to a class of postdominance relations that do not necessarily reduce to a tree.

4 Nontermination (In-)Sensitive Control Dependence

But remember not a game new under the sun
Everything you did has already been done
I know all the tricks from bricks to kingston

(Lauryn Hill — Lost Ones)

In [Ran+07], the authors introduce both **n**on-**t**erminiation **s**ensitive and -**i**nsensitive notions of **c**ontrol **d**ependence ($\rightarrow_{\text{ntscd}}$ and $\rightarrow_{\text{nticd}}$) suitable for *modern program structures*. Specifically, they develop these notions for control flow graphs that do not have a unique exit node n_x. They also propose suitable notions of correctness for nontermination sensitive slicing, and give proof of correspondence for slices involving $\rightarrow_{\text{ntscd}}$. They do not, however, provide a notion of slicing correctness for nontermination insensitive slicing, or for $\rightarrow_{\text{nticd}}$. Instead in [Amt08], the authors need a new kind of nontermination insensitive dependence. In general, neither $\rightarrow_{\text{ntscd}}$ nor $\rightarrow_{\text{nticd}}$ slices are sound:

- Sound nontermination sensitive slices must be closed not only under $\rightarrow_{\text{ntscd}}$ and data dependence, but also under ***d**ecisive **o**rder **d**ependence* ([Ran+07]).
- Sound nontermination *in*sensitive slices must be closed not under $\rightarrow_{\text{nticd}}$ and data dependence, but instead under ***w**eak **o**rder **d**ependence* and data dependence ([Amt08]).

The authors claim algorithms for the computation of nontermination sensitive and insensitive control dependence, and decisive order dependence, all with running time polynomial in the size of the control flow graph.

In this chapter, I merely repeat definitions and basic results from [Ran+07]. However for those interested, a detailed treatment of the algorithms from [Ran+07] can be found in Appendix B. There in Section B.1, I explain why their algorithms for $\rightarrow_{\text{ntscd}}$ and $\rightarrow_{\text{nticd}}$ are flawed. In particular, there is no immediately obvious way to repair the algorithm for $\rightarrow_{\text{nticd}}$. In order to find a correct algorith for $\rightarrow_{\text{nticd}}$, in Section B.2 I try to explain how nontermination-sensitive control dependence naturally corresponds to *least* fixed points, while nontermination-insensitive control dependence corresponds to *greatest* fixed points. I do this by an analogy with *liveness* and *safety* properties. Indeed in Section B.3, I propose a fixed-point based characterization for $\rightarrow_{\text{ntscd}}$ and $\rightarrow_{\text{nticd}}$ that immediately yields correct algorithms both for $\rightarrow_{\text{ntscd}}$ and $\rightarrow_{\text{nticd}}$. There, $\rightarrow_{\text{nticd}}$ and $\rightarrow_{\text{ntscd}}$ are obtained from least and greatest fixed point, respectively, of the *same* functional, yielding the informal slogan:

$$\begin{aligned} \text{nontermination}\quad \text{sensitivity} &= \text{least fixed point} \\ \text{nontermination insensitivity} &= \text{greatest fixed point} \end{aligned}$$

4.1 Nontermination (In-)Sensitive Control Dependence in Arbitrary Graphs

Standard postdominance $\sqsupseteq_{\text{POST}}$ requires a unique exit node n_x to formalize the notion of a node m always "eventually" executing after an execution of n. In order to formalize this notions for arbitrary graphs (possibly lacking a unique exit node n_x), in [Ran+07] the authors introduce the notion of *control sinks*.

Definition 4.1.1. Mostly following [Ran+07], I define:

- A path π is called *maximal* if it is infinite, or its last node is an exit node (i.e.: it has no outgoing edges). With

  $${}_{n}\Pi^{G}_{\text{MAX}} = \{\pi \mid \pi = n, \ldots \text{ and } \pi \text{ is maximal}\}$$

  I denote the set of maximal paths starting in n.

- A set $S \subseteq \text{N}$ is called a *control sink* if it is a strongly connected component of G, and successors y of nodes $x \in S$ remain in S:

  $$\frac{x \to_G y \qquad x \in S}{y \in S}$$

- A control sink is called trivial if it consists of one node. This is then either a (not necessarily unique) exit node, or a node with a self-edge (and no other outgoing edges).

- A maximal path π is called *sink-bound* (for some control-sink S) if

  - there is some node $n_s \in \pi \cap S$
  - if S is non-trivial, then all nodes in S appear in π infinitely often.

I also call such paths simply *sink paths*. With

$$_{n}\Pi^{G}_{\text{SINK}} = \{\pi \mid \pi = n, \ldots \text{ and } \pi \text{ is sink-bound}\}$$

I denote the set of sink paths starting in n.

- The set of *conditional nodes* of G is

$$\text{COND}_G = \{n \mid \exists n_l \neq n_r.\ n \rightarrow_G n_l \ \wedge\ n \rightarrow_G n_r\}$$

Definition 4.1.2. Let $m, n \in N$ be nodes in a graph G. Then m is said to be *non-termination sensitively/insensitively control-dependent on n*, written $n \rightarrow_{\text{ntscd}} m$ / $n \rightarrow_{\text{nticd}} m$, if there exists edges

$$n \rightarrow_G n_l, \quad n \rightarrow_G n_r$$

such that all maximal/sink- paths from n_l contain m, but not all maximal/sink- paths from n_r do:

$n \rightarrow_{\text{ntscd}} m$ iff

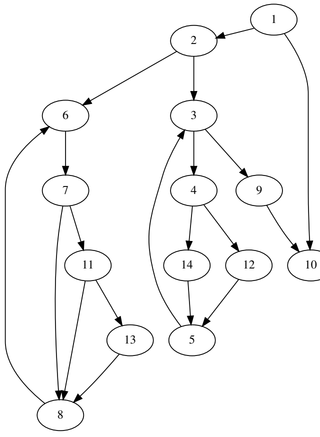
- $\forall \pi \in {}_{n_l}\Pi_{\text{MAX}}.\ m \in \pi$
- $\neg\forall \pi \in {}_{n_r}\Pi_{\text{MAX}}.\ m \in \pi$

$n \rightarrow_{\text{nticd}} m$ iff

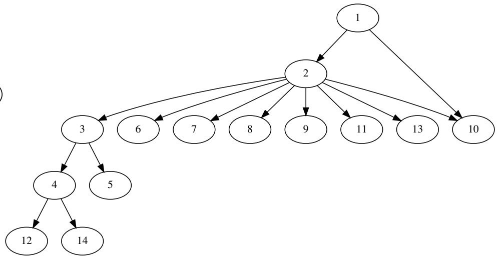
- $\forall \pi \in {}_{n_l}\Pi_{\text{SINK}}.\ m \in \pi$
- $\neg\forall \pi \in {}_{n_r}\Pi_{\text{SINK}}.\ m \in \pi$

In Figure 4.1b, I show nontermination insensitive control dependence for the graph G in Figure 4.1a. It is nontermination insensitive because, for example, node 9 is *not* dependent on node 3, which can in principle delay the execution of node 9 infinitely.

The graph G has one non-trivial control sink $\{6, 7, 8, 11, 13\}$. Note that every node in this sink is control-dependent on node 2, but within this sink, there are *no* control dependencies. Intuitively, one might have expected, for example, node 11 to be dependent on node 7. Although node 7 can prevent node 11 from being executed in principle, it can only do so by delaying it infinitely (i.e.: by always choosing the successor 8). But nontermination-insensitivity here causes $\rightarrow_{\text{nticd}}$ to ignore such executions. The fact that there are no such $\rightarrow_{\text{nticd}}$-dependencies within control-sinks means that $\rightarrow_{\text{nticd}}$ alone cannot be used for slicing in information-flow applications.

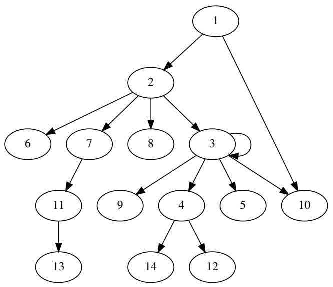


(a) A CFG G



(b) $\rightarrow^{G}_{\text{nticd}}$

Figure 4.1: Nontermination Insensitive Control Dependence



(a) $\rightarrow^{G}_{\text{ntscd}}$

Figure 4.2: Nontermination Insensitive Control Dependence

In Figure 4.2a, I show nontermination sensitive control dependence for the same graph CFG from Figure 4.1a. It is sensitive because, for example, node 9 *is* dependent on node 3, which can delay the execution of node 9 infinitely. Also, node 11 *is* control dependent on node 7. In this graph G, $\rightarrow_{\text{ntscd}}$ induces sound slices, but this is not so for every graph G.

Definition 4.1.3. In the words of Definition 3.1.2 from Section 3.2, $\rightarrow_{\text{nticd}}$ is just $\sqsupseteq_{\text{SINK}}$-control dependence, and $\rightarrow_{\text{ntscd}}$ is just $\sqsupseteq_{\text{MAX}}$-control dependence, where

$$\begin{aligned} m \sqsupseteq^G_{\text{SINK}} n \quad &\Leftrightarrow \quad \forall \pi \in {}_n\Pi^G_{\text{SINK}}.\ m \in \pi \\ m \sqsupseteq^G_{\text{MAX}} n \quad &\Leftrightarrow \quad \forall \pi \in {}_n\Pi^G_{\text{MAX}}.\ m \in \pi \end{aligned}$$

I say that $\sqsupseteq^G_{\text{SINK}}$ is a nontermination *insensitive* form of postdominance, while $\sqsupseteq^G_{\text{MAX}}$ is a nontermination *sensitive*.

Nontermination insensitive control dependence $\rightarrow_{\text{nticd}}$ generalizes standard control dependence to graphs without unique exit node n_x. Nontermination sensitive control dependence $\rightarrow_{\text{ntscd}}$ generalizes *weak* control-dependence [PC90] to graphs without unique exit node. Intuitively, the difference between $\rightarrow_{\text{nticd}}$ and $\rightarrow_{\text{ntscd}}$ is that $\rightarrow_{\text{nticd}}$ assumes "every loop to terminate". Thus I have $n \rightarrow_{\text{ntscd}} m$ even if the only possibility of conditional node n to prevent the execution of m is by infinitely often choosing the same successor n_l. On the other hand, $\rightarrow_{\text{nticd}}$ assumes such loops to terminate, i.e.: it expects n to eventually choose a *different* successor n_r.

Indeed, Ranganath et al. give proofs for the following properties:

Theorem 4.1.1 (Ranganath et al., Theorem 1)**.** Let G have a unique exit node n_x. Then

$$n \rightarrow_{\text{nticd}} m \iff n \rightarrow_{\text{cd}} m$$

Theorem 4.1.2 (Ranganath et al., Theorem 4)**.** Let G be *any* CFG.Then

$$n \rightarrow_{\text{nticd}} m \implies n \rightarrow_{\text{ntscd}} m$$

Summary

- Nontermination sensitive and insensitive control dependence generalize standard control dependence notions to graphs without unique exit node.
- But these notions alone do not lead to sound slices.
- They can be defined uniformly, by reference to nontermination sensitive and insensitive postdominance $\sqsupseteq_{\mathrm{MAX}}$ and $\sqsupseteq_{\mathrm{SINK}}$.

5 Postdominator Pseudoforests

You come to me for advice, but you can't cope with anything you don't recognize. Hmmm. So we'll have to tell you something you already know but make it sound like news, eh. Well, business as usual , I suppose.

(Douglas Adams — The Ultimate Hitchhiker's Guide to the Galaxy)

Neither the (originally flawed) Algorithm 14 for $\rightarrow_{\text{ntscd}}$ due to [Ran+07], nor my correct algorithm Algorithm 16 for $\rightarrow_{\text{nticd}}$ from Appendix B offer performance comparable to the efficient-in-practice algorithm for standard control dependence $\rightarrow_{\text{cd}}$.

But with the algorithm for generalized control dependence from Section 3.2, I can immediately derive efficient-in-practice algorithm for both $\rightarrow_{\text{nticd}}$ and $\rightarrow_{\text{ntscd}}$ if

1. $\sqsupseteq_{\text{SINK}}$ and $\sqsupseteq_{\text{MAX}}$ admit efficient PDF partitionings, and
2. I can provide efficient-in-practice algorithm for the computation of some transitive reduction $<$ of $\sqsupseteq_{\text{SINK}}$ and $\sqsupseteq_{\text{MAX}}$.

In order to do this, I first give in Section 5.1 fixed-point characterizations of $\sqsupseteq_{\text{SINK}}$ and $\sqsupseteq_{\text{MAX}}$. Then in Section 5.2 and Section 5.3 I generalize the notion of standard postdominator trees to obtain the new notion of postdominator *pseudo-forests*. These will materialize as transitive reductions of nontermination sensitive and insensitive postdominance $\sqsupseteq_{\text{MAX}}$ and $\sqsupseteq_{\text{SINK}}$, and together with the fixed-point characterizations of $\sqsupseteq_{\text{SINK}}$ and $\sqsupseteq_{\text{MAX}}$ immediately lead to efficient-in-practice algorithms for their computation.

For the nontermination sensitive case $\rightarrow_{\text{ntscd}}$, I am not aware of previously existing efficient algorithms even for graphs with unique exit node n_x.[1] For the nontermination insensitive case, on the other hand, such algorithms exist ([LT79; Cyt+91]). Indeed if I were only interested in ("offline") algorithms for $\rightarrow_{\text{nticd}}$, I would not need the development in Section 5.3, because there exists a reduction $G \mapsto G_S$ such that G_S is a graph with unique-exit node, and nontermination insensitive control dependence $\rightarrow^{G}_{\text{nticd}}$ in G can easily be obtained from standard control-dependence $\rightarrow^{G_S}_{\text{cd}}$ in G_S. I explain this reduction in Section 5.4. Later in Section 6.7 however, I *will* make use of the algorithms developed in Section 5.3 in order to support an *incremental* recomputation of a series $(\sqsupseteq^{G_m}_{\text{SINK}})_{m \in M}$ of postdominance pseudo-forests $\sqsupseteq^{G_m}_{\text{SINK}}$ for a sequence $(G_m)_{m \in M}$ of related graphs.

[1] In [BP96], the authors provide an efficient algorithm for the computation of the *Augmented Loop-Postdominator-Tree*, which for CFG with unique exit nodes is derived by augmenting a nontermination-sensitive postdomominance forest. Using this tree, they give an algorithm to answer weak (i.e.: nontermination-sensitive) control dependence *queries*, but do not give an explicit algorithm to efficiently compute the whole weak control relation.

5.1 Fixed-Point Characterizations for Postdominance

It is implicit already in, e.g., [HU73], that $\sqsupseteq_{\mathrm{POST}}$ (i.e.: the postdominance relation underlying standard control dependence for graphs with unique end node n_x) has a *greatest* fixed point characterization:

Theorem 5.1.1. Let G be a CFG with unique end node n_x and P be the rule-system

$$\frac{}{n \sqsupseteq n}\,\mathsf{P}^{\mathrm{self}} \qquad \frac{\forall p \to_G x.\ m \sqsupseteq x \qquad p \neq n_x}{m \sqsupseteq p}\,\mathsf{P}^{\mathrm{suc}}$$

Then $\sqsupseteq_{\mathrm{POST}} = \nu\mathsf{P}$ over the lattice $(N \times N, \subseteq)$.

Proof: On page 354 in the appendix. □

Note that when computing $\sqsupseteq_{\mathrm{POST}}$ as the greatest fixed point of P, fixed point iteration starts with the initial value $\top = N \times N$, i.e. with the assumption

$$m \sqsupseteq n \text{ for } \textit{all } m, n$$

regardless whether m is even reachable in G from n. This is correct because an assumption $m \sqsupseteq n$ for $\neg\, n \to_G^* m$ must be validated by $m \sqsupseteq x$ for all G-successors x of n. But eventually I must reach n_x *without* ever reaching m before, and at n_x i cannot validate $m \sqsupseteq n_x$.

Usually, P is presented as a system of equations:

$$\begin{array}{lcll} \mathrm{ipdom}_{\mathrm{POST}}(n_x) & = & \{n_x\} & \\ \mathrm{ipdom}_{\mathrm{POST}}(n) & = & \{n\ \} \cup \bigcap_{n \to_G x} \mathrm{ipdom}_{\mathrm{POST}}(x) & n \neq n_x \end{array}$$

An algorithm loosely based on P is given in [CHK01]. Here, the authors — while computing *immediate* postdominators — do *not* explicitly (or implicitly!) start with the assumption that any node m may post-dominate any node n. Instead, they make crucial use of the fact that any preliminary *guess* $\text{ipdom}(n) = x$ for any G-successor x of n is always O.K. in the sense that x will at least always *reach* the *true* $\text{ipdom}(n)$ (by following ipdom upwards the dominator tree).

Following the idea that $\rightarrow_{\text{nticd}}$ is a generalization of standard control dependence (i.e.: $\sqsupseteq_{\text{POST}}$ control dependence), and $\sqsupseteq_{\text{SINK}}$ a generalization of $\sqsupseteq_{\text{POST}}$, I must suspect that the greatest fixed point of *some* generalization of P describes $\sqsupseteq_{\text{SINK}}$ for CFGs without unique end nodes. Moreover, the slogan

$$\begin{aligned} \text{non-termination insensitivity} &= \text{greatest fixed point} \\ \text{non-termination sensitivity} &= \text{least fixed point} \end{aligned}$$

then also suggests that it's *least* fixed point characterizes $\sqsupseteq_{\text{MAX}}$. In fact:

Theorem 5.1.2. Let G be any CFG and D be the rule-system

$$\frac{}{n \sqsupseteq n}\,\mathsf{D}^{\text{self}} \qquad \frac{\forall p \rightarrow_G x.\ m \sqsupseteq x \qquad p \rightarrow_G^* m}{m \sqsupseteq p}\,\mathsf{D}^{\text{suc}}$$

Then

$$\begin{aligned} \nu\mathsf{D} &= \sqsupseteq_{\text{SINK}} \\ \mu\mathsf{D} &= \sqsupseteq_{\text{MAX}} \end{aligned}$$

Proof: On page 354 in the appendix. □

Remark 5.1.1. A formal proof of Theorem 5.1.2 checked by the Isabelle/HOL proof assistant is available from Simon Bischof [Bis19].

When generalizing to D from P, I had to add the reachability constraint $p \rightarrow_G^* m$, since otherwise I could — for example: given some control sink S, and nodes $n \in S$ but $m \notin S$ — validate $m \sqsupseteq_{\text{SINK}} n$

merely by mutual application of rule $\mathsf{D}^{\mathrm{suc}}$ (to all nodes in S), even if m is not even reachable from n. Given this reachability constraint, I do not need a generalization of the constraint $p \neq n_x$, since it's main function was to prevent application of rule $\mathsf{P}^{\mathrm{suc}}$ to nodes p that do not have *any* successors (of which n_x is the only one in CFG with unique exit node). But in D, this is already accomplished by the reachability constraint $p \rightarrow_G^* m$.

Remark 5.1.2. Rule system D can be read as a simplified version of the following rule system D_3 from the appendix.

$$\frac{m \in \overline{\mathrm{next}_{\mathrm{COND}}}\,[x]}{m \sqsupseteq x}\,\mathsf{D}_3^{\mathrm{lin}} \qquad \frac{\begin{array}{c} n = \mathrm{next}_{\mathrm{COND}}\,[x] \\ \forall n \rightarrow_G x.\, m \sqsupseteq x \qquad x \rightarrow_G^* m \end{array}}{m \sqsupseteq x}\,\mathsf{D}_3^{\mathrm{cond}}$$

But D_3, then, can be read as just a variant of the rules S_3 (Definition B.2.5 in the appendix).

Both D fixed-points can be computed naively, using an explicit representations of the relation $\sqsupseteq$. For more efficient algorithms, I need instead a sparse representations of $\sqsupseteq$, which is available both for $\sqsupseteq_{\mathrm{MAX}}$ and $\sqsupseteq_{\mathrm{SINK}}$ in form of *pseudo forests*.

5.2 Nontermination Sensitive Pseudoforests

Before I devise an efficient algorithm for nontermination sensitive postdominance $\sqsupseteq_{\mathrm{MAX}}$, let me first affirm that, indeed, $\sqsupseteq_{\mathrm{MAX}}$ admits efficient PDF partitionings, as required if I want to use my generalized postdominance frontiers algorithm from Section 3.2. It is easy to see that the first requirement is met:

Lemma 5.2.1. $\sqsupseteq_{\mathrm{MAX}}$ is reflexive and transitive.

Proof (Sketch): By definition.

In order to establish he remaining requirements, I investigate the structure of $\sqsupseteq_{\mathrm{MAX}}$.

Observation 5.2.1. Let G be any CFG, and $>_{\mathrm{MAX}}$ any transitive reduction of $\sqsupseteq_{\mathrm{MAX}}$. Then the graph

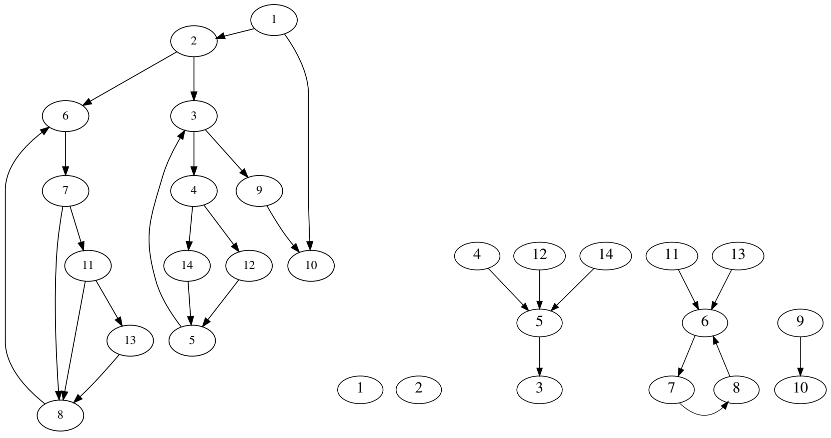
$$(N, <_{\mathrm{MAX}}) \;=\; (N, \{\, (m, n) \mid n >_{\mathrm{MAX}} m \,\})$$

is a *pseudo forest*, i.e.: a graph with at most one successor at each node n.

A glance at Figure 5.1b will justify the name *pseudo forest*: Visible are five independent pseudo-trees (where roots sometimes consist of *multiple* nodes $n \in N$) with roots: $1, 2, 3, \{6, 7, 8\}, 10$. The roots are shown near the bottom, and arrows $\rightarrow$ follow $<_{\mathrm{MAX}}$.

Observation 5.2.2. Let $>_{\mathrm{MAX}}$ be any transitive reduction of $\sqsupseteq_{\mathrm{MAX}}$. Then

$$\begin{array}{rcl} x \;1\text{-}\!\sqsupseteq_{\mathrm{MAX}}\; z & \Longleftrightarrow & x >^{+}_{\mathrm{MAX}} z \\ x \in \mathrm{ipdom}_{\sqsupseteq_{\mathrm{MAX}}}(z) & \Longleftrightarrow & x' >^{*}_{\mathrm{MAX}} x >^{*}_{\mathrm{MAX}} x' \\ & & \text{for some } x' \text{ s.t. } x' >_{\mathrm{MAX}} z \end{array}$$



(a) A CFG

(b) A transitive reduction $<_{\text{MAX}}$ of it's relation $\sqsupseteq_{\text{MAX}}$

Figure 5.1: A Dominator Pseudo-Forest

and consequently:

$$\text{ipdom}^*_{\sqsupseteq_{\text{MAX}}} \quad = \quad \sqsupseteq_{\text{MAX}}$$

Lemma 5.2.2. $\sqsupseteq_{\text{MAX}}$ lacks joins.

$$\frac{\begin{array}{ccc} x \in \text{ipdom}_{\sqsupseteq_{\text{MAX}}}(v) & v \sqsupseteq_{\text{MAX}} s & \\ & & z \neq v \\ x \in \text{ipdom}_{\sqsupseteq_{\text{MAX}}}(z) & z \sqsupseteq_{\text{MAX}} s & \end{array}}{v \in \text{ipdom}_{\sqsupseteq_{\text{MAX}}}(z) \ \vee z \in \text{ipdom}_{\sqsupseteq_{\text{MAX}}}(v)}\ \text{NoJoin}$$

Proof: On page 355 in the appendix. □

Lemma 5.2.3. $\sqsupseteq_{\text{MAX}}$ is closed under $\rightarrow_G$.

$$\frac{y \rightarrow_G x \qquad x' \sqsupseteq_{\text{MAX}} y \qquad x' \neq y}{x' \sqsupseteq_{\text{MAX}} x}\,\text{CL}^{\rightarrow_G}$$

Proof: By definition. □

The Algorithm for computing a pseudo-forest $<_{\text{MAX}}$ will require the computation of *least common ancestors* for (preliminary) pseudo-forests:

Definition 5.2.1. Given a pseudo-forest $<$ and two nodes n, m

$$\text{lca}_<(n, m) = \left\{ a \;\middle|\; \begin{array}{c} n <^* a \;\wedge\; m <^* a \\ \forall a'. \;\; (n <^* a' \wedge m <^* a') \implies a <^* a' \end{array} \right\}$$

More generally: for any set S of nodes:

$$\text{lca}_<(S) = \left\{ a \;\middle|\; \begin{array}{c} \forall n \in S.\, n <^* a \\ \forall a'. \;\; (\forall n \in S.\, n <^* a') \implies a <^* a' \end{array} \right\}$$

A least common ancestor $\text{lca}_<(S)$ of a set S can be computed by iterating over nodes $n \in S$:

Proposition 5.2.1. Given a pseudo-forest $<$ and nodes x, y, z, if $a_{xy} \in \text{lca}_<(x, y)$ and $a_{xyz} \in \text{lca}_<\left(a_{xy}, z\right)$, then $a_{xyz} \in \text{lca}_<(\{x, y, z\})$.

The least common ancestor can naively be computed by, e.g., Algorithm 3, and be extended to sets via Algorithm 4. See Algorithm 18 in Appendix D for the slightly more performant variant used in all benchmarks.

```
Input   : A pseudo-forest $<$, represented as a map
          $\mathsf{IMDOM} : N \hookrightarrow N$ s.t. $\mathsf{IMDOM}\,[n] = m$ iff $n < m$.
Input   : Nodes $\mathsf{m}_0,\ \mathsf{n}_0$
Output  : A least common ancestor of $\mathsf{n}_0, \mathsf{m}_0$, or $\bot$ if there is
          none.
begin
    return $\mathsf{lca}\,(\mathsf{n}_0,\ \mathsf{m}_0)$
end
Function $\mathsf{lca}\,(\pi_{\mathsf{n}},\ \pi_{\mathsf{m}})$
    Input   : A $<$-path $\pi_{\mathsf{n}} = \mathsf{n}_0, \ldots, \mathsf{n}$ ending in $\mathsf{n}$
    Input   : A $<$-path $\pi_{\mathsf{m}} = \mathsf{m}_0, \ldots, \mathsf{m}$ ending in $\mathsf{m}$
    if $\mathsf{m} \in \pi_{\mathsf{n}}$ then
        return $\mathsf{m}$
    end
    if $\mathsf{n} \in \pi_{\mathsf{m}}$ then
        return $\mathsf{n}$
    end

    switch $\{\mathsf{IMDOM}[\mathsf{n}]\} \setminus \pi_{\mathsf{n}}$ do
        case $\emptyset$ do
            switch $\{\mathsf{IMDOM}[\mathsf{m}]\} \setminus \pi_{\mathsf{m}}$ do
                case $\emptyset$ do
                    return $\bot$
                end
                case $\{\mathsf{m'}\}$ do
                    return $\mathsf{lca}(\pi_{\mathsf{m}}\mathsf{m'},\ \pi_{\mathsf{n}})$
                end
            end
        end
        case $\{\mathsf{n'}\}$ do
            return $\mathsf{lca}(\pi_{\mathsf{m}},\ \pi_{\mathsf{n}}\,\mathsf{n'})$
        end
    end
end
```

Algorithm 3: A *least common ancestor* algorithm.

```
Input  : A pseudo-forest $<$, represented as a map $\mathsf{IMDOM} : N \hookrightarrow N$
         s.t. $\mathsf{IMDOM}[n] = m$ iff $n < m$.
Input  : A set $\{\mathsf{n}\} \cup \mathsf{S}$ of at least one Node
Output: A least common ancestor of $\{\mathsf{n}\} \cup \mathsf{S}$, or $\bot$ if there is none.
begin
    $\mathsf{a} \leftarrow \mathsf{n}$
    for $s \in S$ do
        $\mathsf{a} \leftarrow \mathsf{lca}(\mathsf{a}, \mathsf{s})$
    end
end
```

Algorithm 4: A *least common ancestor* algorithm for sets.

5.2.1 An Algorithm for Nontermination Sensitive Postdominance

I can now present Algorithm 5 for the computation of some transitive reduction $>_{\text{MAX}}$ of $\sqsupseteq_{\text{MAX}}$. It can be understood as a least fixed point computation (of $\mu\mathsf{D}_3$) using an *efficient representation* $<_{\text{MAX}} \in \alpha(\sqsupseteq_{\text{MAX}})$ of $\sqsupseteq_{\text{MAX}}$. Informally:

$$\begin{aligned}\gamma(<) &= <^* \\ \alpha(\sqsupseteq) &= \{ < \mid <^* = \sqsubseteq \}\end{aligned}$$

with (abstract) rule system $\mathsf{D}_3^\#$

$$\frac{m \rightarrow_G x \qquad \neg m \in \text{COND}}{m <_{\text{MAX}} x}$$

$$\frac{a = \epsilon\, a.\; a \in \mathrm{lca}_{<_{\text{MAX}}}(\{ x \mid m \rightarrow_G x \}) \qquad m \in \text{COND} \qquad m \rightarrow_G^* a}{m <_{\text{MAX}} a}$$

In order to guarantee termination, the algorithm will choose

$$\epsilon\ a.\ a \in \mathrm{lca}_{<_{\text{MAX}}}(\dots)$$

consistently (i.e.: always choose the same a for a given input.) The Algorithm 5 is a straight-forward implementation of a chaotic least fixed point iteration. The update of workset can be implemented reasonably efficiently if the set $\{y \mid y <^* x\}$ can be enumerated efficiently, which can, for example, be done by maintaining a second data structure $\mathsf{IMDOM} : N \to 2^N$ representing $>$.

Observation 5.2.3. Let G be any CFG. Then Algorithm 5 terminates with a result $<_{\text{MAX}}$ s.t. $>_{\text{MAX}}$ is a transitive reduction of $\sqsupseteq_{\text{MAX}}$.

Also see Subsection A.2.1 in the appendix for a sketch a proof of Observation 5.2.3.

This chaotic iteration can be improved slightly by making use of the fact that for each node $x \in \text{COND}$, the value $\mathsf{IMDOM}[x]$ is never changed from some z_0 back to $\perp$, and changed from some z_0 to some z only for $z_0, z \in \mathrm{ipdom}_{\sqsupseteq_{\text{MAX}}}(x)$: never re-insert nodes n s.t. $\mathsf{IMDOM}[n] \neq \perp$.

For the very same reason, a sequential (non-chaotic) variant of this least fixed point computation can be implemented reasonably efficiently: The resulting algorithm (Algorithm 19 in Appendix D) maintains a FIFO queue which at all times contains those nodes $x \in \text{COND}$ for which $\mathsf{IMDOM}[x] = \perp$, re-inserting nodes x only if no $\mathsf{lca} \neq \perp$ was found. It terminates as soon as the whole queue was traversed once without it becoming smaller.

```
Input   : A CFG $G$
Data: A pseudo-forest $<$ represented as a map $\mathsf{IMDOM} : N \hookrightarrow N$ s.t.
      $\mathsf{IMDOM}\,[n] = m$ iff $n < m$
Output: A transitive reduction $<_{\text{MAX}}$ of $\sqsupseteq_{\text{MAX}}$
begin
    for $x \in N,\ \{z \mid x \to_G z\} = \{z\},\ z \neq x$ do
        $\mathsf{IMDOM}\,[x] \leftarrow z$
    end
    MAXIMAL$_{\text{up}}$
    return $\mathsf{IMDOM}$
end
Procedure MAXIMAL$_{\text{up}}$
    $\mathsf{workset} \leftarrow \text{COND}_G$
    while $\mathsf{workset} \neq \emptyset$ do
        $x \leftarrow remove(\mathsf{workset})$
        $a \leftarrow \mathsf{lca}\,(\{\, y \mid x \to_G y \,\})$
        $z \leftarrow \begin{cases} \bot & \text{if } a = \bot \vee a = x \\ a & \text{otherwise} \end{cases}$
        assert $z \neq \mathsf{IMDOM}[x] \Rightarrow z \neq \bot$
        assert $z \neq \mathsf{IMDOM}[x] \wedge \mathsf{IMDOM}[x] = z_0 \Rightarrow z_0 <^* z <^* z_0$
        if $z \neq \mathsf{IMDOM}[x]$ then
            $\mathsf{workset} \leftarrow \mathsf{workset} \cup \{n \in \text{COND} \mid n \neq x, \exists n \to_G y.\, y <^* x\}$
            $\mathsf{IMDOM}\,[x] \leftarrow z$
        end
    end
end
```

Algorithm 5: An algorithm for the computation of some $<_{\text{MAX}}$.

5.3 Nontermination Insensitive Pseudoforests

Just as Algorithm 5 for a transitive reduction $>_{\text{MAX}}$ of $\sqsupseteq_{\text{MAX}}$ corresponds to a computation of the least fixed point $\mu\mathsf{D}$, the algorithm for a transitive reduction $>_{\text{SINK}}$ of $\sqsupseteq_{\text{SINK}}$ will resemble a *greatest* fixed point computation (of the functional D). The key of obtaining an efficient algorithm will be

1. To avoid explicit reachability checks $m \rightarrow_G^* a$. These were unnecessary in the least fixed point computation because there, reachability was established during the iteration starting from $\bot = \emptyset \subseteq N \times N$. In the greatest fixed-point computation, however, these would become necessary if I were to start a greatest fixed point computation at $\top = N \times N$.
2. To instead find a *pseudo forest* $<$ such that

$$>^* \quad \supseteq \quad \sqsupseteq_{\text{MAX}}$$

to start the fixed point iteration from.

I first affirm that $\sqsupseteq_{\text{SINK}}$ does, indeed, admit efficient PDF partitionings. Again, it is easy to see that the first requirement is met:

Lemma 5.3.1. $\sqsupseteq_{\text{SINK}}$ is reflexive and transitive.

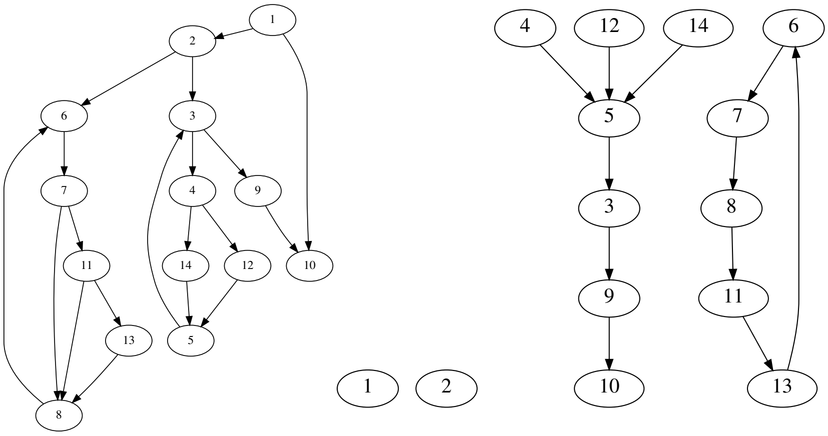
Proof (Sketch): By definition.

Also, the structure of transitive reductions $>_{\text{SINK}}$ is not different from before:

Observation 5.3.1. Let G be any CFG, and $>_{\text{SINK}}$ any transitive reduction of $\sqsupseteq_{\text{SINK}}$. Then the graph

$$(N, <_{\text{SINK}}) \;=\; (N, \{\, (m, n) \;\mid n >_{\text{SINK}} m \,\})$$

is a *pseudo forest*.



(a) A CFG

(b) A transitive reduction $<_{\text{SINK}}$ of it's relation $\sqsupseteq_{\text{SINK}}$

Figure 5.2: A Dominator Pseudo Forest

In Figure 5.2b such a pseudo forest $<_{\text{SINK}}$ is shown for the previously used example CFG. Visible are four independent pseudo-trees with roots: $1, 2, 10, \{6, 7, 8, 11, 13\}$.

Observation 5.3.2. Let $<_{\text{SINK}}$ be any transitive reduction as above. Then

$$\begin{aligned} x \ 1\text{-}\sqsupseteq_{\text{SINK}} z &\iff x >^{+}_{\text{SINK}} z \\ x \in \text{ipdom}_{\sqsupseteq_{\text{SINK}}}(z) &\iff x >^{*}_{\text{SINK}} x' >^{*}_{\text{SINK}} x \\ & \text{for some } x' \text{ s.t. } x' >_{\text{SINK}} z \end{aligned}$$

and consequently:

$$\text{ipdom}^{*}_{\sqsupseteq_{\text{SINK}}} \quad = \quad \sqsupseteq_{\text{SINK}}$$

Just as before, $\sqsupseteq_{\text{SINK}}$ lacks joins, and is closed under $\to_G$:

Observation 5.3.3.

$$\frac{\begin{array}{ccc} x \in \mathrm{ipdom}_{\sqsupseteq_{\mathrm{SINK}}}(v) & v \sqsupseteq_{\mathrm{SINK}} s & \\ x \in \mathrm{ipdom}_{\sqsupseteq_{\mathrm{SINK}}}(z) & z \sqsupseteq_{\mathrm{SINK}} s & z \neq v \end{array}}{v \in \mathrm{ipdom}_{\sqsupseteq_{\mathrm{SINK}}}(z) \ \lor z \in \mathrm{ipdom}_{\sqsupseteq_{\mathrm{SINK}}}(v)} \text{NoJoin}$$

Proof: Just as for Lemma 5.2.2. □

Observation 5.3.4.

$$\frac{y \to_G x \qquad x' \sqsupseteq_{\mathrm{SINK}} y \qquad x' \neq y}{x' \sqsupseteq_{\mathrm{SINK}} x} \mathrm{CL}^{\to_G}$$

Additionally, the sinks of the CFG determine $<_{\mathrm{SINK}}$ to some degree:

Observation 5.3.5. Let $>_{\mathrm{SINK}}$ be a transitive reduction of $\sqsupseteq_{\mathrm{SINK}}$. Then the multi-node roots of the pseudo-forest $<_{\mathrm{SINK}}$ are exactly the multi-node sinks of G. Also, any single-node sink of G is a single-node root in $<_{\mathrm{SINK}}$.

The Algorithm 6 for the computation of $<_{\mathrm{SINK}}$ then works in two phases. In the first phase $\mathrm{SINK}_{\mathrm{up}}$, it computes some approximation $<_0$ "just above" $<_{\mathrm{SINK}}$, i.e.: a pseudo-forest $<_0$ such that

$$>_0^* \quad \supseteq \quad \sqsupseteq_{\mathrm{SINK}}$$

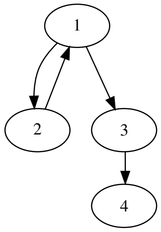
The phase $\mathrm{SINK}_{\mathrm{down}}$ then computes $<_{\mathrm{SINK}}$ by (greatest) fixed-point iteration from above.
In $\mathrm{SINK}_{\mathrm{up}}$, I necessarily have to be more lenient than I was in $\mathrm{MAXIMAL}_{\mathrm{up}}$. There, in order to establish $x < a$, I needed positive evidence that $a \in \mathrm{lca}_<(\{ y \mid x \to_G y \})$. Now, even if $\mathrm{lca}_<(\{ y \mid x \to_G y \}) = \emptyset$, I will sometimes have to chose an *assumption* $x < a$ for some reasonable a, which I later either validate, weaken or abandon in $\mathrm{SINK}_{\mathrm{down}}$.

Consider the preliminary pseudo forest $<$ in Figure 5.3b. I need to establish $1 < 3$ (i.e.: that 3 is an "immediate postdominator" of 1), but I find that $\mathrm{lca}_{<}(\{2,3\}) = \emptyset$. Now I would *like* to assume *both*, $1 < 3$ and $1 < 2$, the latter of which would then be invalidated in phase $\mathrm{SINK}_{\mathrm{down}}$. But then $<$ no longer would be a pseudo forest! If I were to assume just $1 < 2$, I would obtain a $>_0$ to start the phase $\mathrm{SINK}_{\mathrm{down}}$ with such that *not*: $>_0^* \supseteq\ \sqsupseteq_{\mathrm{MAX}}$, so I need to make the assumption $1 < 3$. But what is a generally applicable criterion to decide which assumption to make?

Observation 5.3.6. Let $>_{\mathrm{SINK}}$ be a transitive reduction of $\sqsupseteq_{\mathrm{SINK}}$. Then whenever $x <_{\mathrm{SINK}} y$, there is at most one sink S such that any path starting in x is bound for S, and any such S is the same sink that y is bound for.

Note that in this example, for node 3 I have already established $3 <^* 4$ for the sink node $4 \in S$, but I have not yet established $2 <^* 4$ (nor: $2 <^* s$ for *any* sink node s). This suggest that I must — whenever $\mathrm{lca}_{<}(\{\, y \mid x \to_G y \,\}) = \emptyset$ — choose some G-successor node y of x such that already $y <^* s$ for some sink node s. I will call such nodes y *processed*, and maintain in Algorithm 6 a set PROCD of all such nodes. Not shown is the procedure $\mathsf{newProcessed}(x)$ which updates PROCD given a node x s.t. $x <^* s$ for some sink node s, by following linear segments ending in x upwards.

The second phase $\mathrm{SINK}_{\mathrm{down}}$ then corresponds to a (downward) fixed-point iteration that computes of $\nu\mathsf{D}$ starting from $>_0^*$ (instead of: from $\top$). The second phase is essentially the same as the first and only phase $\mathrm{MAXIMAL}_{\mathrm{up}}$ of Algorithm 5. Remember that there, I computed $\mu\mathsf{D}$ from $\bot$. Yet, here are some differences between $\mathrm{SINK}_{\mathrm{down}}$ and $\mathrm{MAXIMAL}_{\mathrm{up}}$. In $\mathrm{MAXIMAL}_{\mathrm{up}}$, I can make use of the fact that for each n, I need to set $\mathsf{IMDOM}[n]$ at most once, while in $\mathrm{SINK}_{\mathrm{down}}$, I may need to update $\mathsf{ISDOM}[n]$ several times. Also, $\mathrm{SINK}_{\mathrm{down}}$ only updates non-sink nodes – sink-nodes were already dealt with during the initialization. Finally, whenever the algorithm would normally set $\mathsf{IMDOM}[x]$ to s for some node s in sink S_i, the algorithm instead sets


(a) A CFG

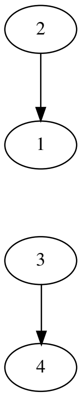

(b) Preliminary pseudo forest $<$

Figure 5.3

$\mathsf{IMDOM}[x]$ to s_i for some canonical sink node $s_i \in S_i$. This is required to ensure termination, since otherwise, the algorithm might oscillate between choosing $x < s$ and $x < s'$ for two nodes $s, s' \in S_i$[2]. Also note that during the workset update with n, here necessarily $n \neq x$.

Dually to $\text{MAXIMAL}_{\text{up}}$, the algorithm $\text{SINK}_{\text{down}}$ can be made slightly more efficient by making use of the fact that once $\mathsf{ISDOM}[x] = \bot$ has been established, it must remain so: never insert such nodes into the workset. Again, chaotic iteration can be replaced with sequential iteration, making use of this same fact.

Observation 5.3.7. Let G be any CFG. Then Algorithm 6 terminates with a result $<_{\text{SINK}}$ s.t. $>_{\text{SINK}}$ is a transitive reduction of $\sqsupseteq_{\text{SINK}}$.

This result concludes Chapter 5 satisfactorily: I have provided algorithms for the computation of both $\sqsupseteq_{\text{MAX}}$ and $\sqsupseteq_{\text{SINK}}$ for *arbitrary* CFG, both of which

[2] In phase $\text{MAXIMAL}_{\text{up}}$ of Algorithm 5, the fact that $<$ approximated $\sqsupseteq_{\text{MAX}}$ was enough to guarantee termination, but here, $<$ approximating $\sqsupseteq_{\text{SINK}}$ is not.

```
Input  : A CFG $G$
Output: A transitive reduction $<_{\text{SINK}}$ of $\sqsupseteq_{\text{SINK}}$
begin
    $\{S_1, \ldots, S_n\} \leftarrow \{S_i \mid S_i \in \mathsf{scc}(G), \neg\, \exists s \rightarrow_G n.\, s \in S_i \,\wedge\, n \notin S_i\}$
     using any efficient scc algorithm.
    $S \leftarrow \bigcup S_i$
    for $1 \leq i \leq n$ do
        $s_i \leftarrow$ any node in $S_i$
        for $n_j \in S$ in any fixed ordering $n_1, \ldots, n_{k_i}$ of $S_i$ do
            $\mathsf{ISDOM}[n_j] \leftarrow n_{j+1 \bmod k_i}$ unless $k_i = 1$
            $\mathsf{processed}(n_j)$
        end
    end
    for $x \in N,\ x \notin S,\ \{z \mid x \rightarrow_G z\} = \{z\},\ z \neq x$ do
        $\mathsf{ISDOM}[x] \leftarrow z$
        if $z \in \mathsf{PROCD}$ then $\mathsf{processed}(x)$
    end

    $\text{SINK}_{\text{up}}$
    $\text{SINK}_{\text{down}}$
    return $\mathsf{ISDOM}$
end
```

Algorithm 6: An algorithm for the computation of some $<_{\text{SINK}}$.

- use efficient data structures (pseudo forests) to compute and represent $\sqsupseteq_{\text{MAX}}$, $\sqsupseteq_{\text{SINK}}$
- do not require explicit reachability checks $m \rightarrow_G^* n$

I derived these algorithm from simple fixed-points characterizations of $\sqsupseteq_{\text{MAX}}$, $\sqsupseteq_{\text{SINK}}$ via D, following the least/greatest fixed point duality.

Later in Chapter 8 validate that these algorithms are efficient in practice.

```
Procedure $\mathrm{SINK}_{\mathrm{up}}$
    workqueue $\leftarrow \mathrm{COND}_G \setminus S$ in any order
    while workqueue $\neq \emptyset$ do
        $x \leftarrow removeFront(\mathsf{workqueue})$
        assert $\mathsf{ISDOM}[x] = \bot \ \wedge \ x \notin \mathsf{PROCD}$
        $\mathsf{SUCCS} \leftarrow \{ y \mid x \rightarrow_G y,\ y \in \mathsf{PROCD} \}$
        if $\mathsf{SUCCS} = \emptyset$ then
            $z \leftarrow \bot$
        else
            $a \leftarrow \mathsf{lca}(\mathsf{SUCCS})$
            $z \leftarrow \begin{cases} \text{any } y \in \mathsf{SUCCS} & \text{if } a = \bot \\ a & \text{otherwise} \end{cases}$
        end
        if $z \neq \bot$ then
            $\mathsf{ISDOM}[x] \leftarrow z$
            $\mathsf{processed}(x)$
        end
        else
            $pushBack(\mathsf{workqueue}, x)$
        end
    end
end
```

Upward Iteration for Algorithm 6.

```
Procedure $\mathrm{SINK}_{\mathrm{down}}$
    workset $\leftarrow \{ n \mid n \in \mathrm{COND}_G \setminus S,\ \mathsf{ISDOM}[n] \neq \bot \}$
    while workset $\neq \emptyset$ do
        $x \leftarrow$ removeMin(workset)
        $a \leftarrow \mathsf{lca}\left(\{ y \mid x \to_G y \}\right)$
        $z \leftarrow \begin{cases} \bot & \text{if } a = \bot \\ s_i & \text{if } a \in S_i \\ a & \text{otherwise} \end{cases}$
        assert $\mathsf{ISDOM}[x] = \bot \Rightarrow z = \bot$
        if $z \neq \mathsf{ISDOM}[x]$ then
            workset $\leftarrow$ workset $\cup \{n \in \mathrm{COND} \setminus S \mid \exists n \to_G y.\ y <^* x\}$
            $\mathsf{ISDOM}[x] \leftarrow z$
        end
    end
end
```
Downward Iteration for Algorithm 6. Set workset is ordered by any fixed ordering of nodes N.

5.4 Reduction to Postdominance Trees

I this section, I present another approach to the computation of $\sqsupseteq_{\text{SINK}}$.

By Observation 5.3.5, the roots (whether trivial or non-trivial) of pseudo-forests $<_{\text{SINK}}$ (i.e.: the $<_{\text{SINK}}$-cycles) are exactly the control-sinks in G, which are exactly the strongly connected components if G that have no outgoing edge. Within such a root (i.e.: within such a set $M \subseteq N$ of nodes), any two nodes $m_1, m_2 \in M$ do $\sqsupseteq_{\text{SINK}}$-postdominate each other, hence no node is ever nontermination insensitively control dependent on any node $m \in M$. At the same time, a node n (possibly not in M) is $\sqsupseteq_{\text{SINK}}$-postdominated by some node $m \in M$ if and only if *every* node $m \in M$ is. In other words: nodes $m \in M$ are equivalent with regard to $\sqsupseteq_{\text{SINK}}$-postdominance. This suggests that it is possible to reduce the computation of $<_{\text{SINK}}$ to a computation a CFG obtained by *condensing* sinks in G.

The Idea behind the following Construction was suggested by Maximilian Wagner:

Observation 5.4.1. Let $G = (N, E)$ be any CFG, and $S_1, \dots, S_n$ its sinks. For $S = \bigcup S_i$, a fresh node $n_x \notin N$, and given for each sink S_i a representative node $s_i \in S_i$, let

$$G_S = (N_S, E_S)$$

$$N_S = \{n_x, s_1, \dots, s_n\} \cup (N \setminus S)$$

$$\begin{aligned} E_S \quad &= \quad \{(n, m) \mid n \rightarrow_G m,\ n, m \notin S\} \\ &\cup \quad \{(n, s_i) \mid n \rightarrow_G s \text{ for some } s \in S_i\} \\ &\cup \quad \{(s_i, n_x) \mid 1 \leq i \leq n\} \end{aligned}$$

Then for nodes $x, m \in N \setminus S$:

$$m \sqsupseteq^{G}_{\text{SINK}} x \quad \Longleftrightarrow \quad m \sqsupseteq^{G_S}_{\text{POST}} x$$

and for $x \in N \setminus S$ and $m \in S_i$:

$$m \sqsupseteq^{G}_{\text{SINK}} x \iff x \in S_i \lor s_i \sqsupseteq^{G_S}_{\text{POST}} x$$

This immediately suggests Algorithm 7 for the computation of some transitive reduction $>_{\text{SINK}}$ of $\sqsupseteq_{\text{SINK}}$

```
Input  : A CFG $G$
Output: A transitive reduction $>_{\text{SINK}}$ of $\sqsupseteq_{\text{SINK}}$ represented as a map
        $\mathsf{ISINKDOM} : N \hookrightarrow N$ s.t. $\mathsf{ISINKDOM}[n] = m$ iff $n <_{\text{SINK}} m$
begin
    $\{S_1, \ldots, S_n\} \leftarrow \{S_k \mid S_k \in \mathsf{scc}(G), \neg \exists s \rightarrow_G n.\ s \in S_k,\ n \notin S_k\}$
      using any efficient scc algorithm.
    $S \leftarrow \bigcup S_i$
    for $1 \le i \le n$ do
        $s_i \leftarrow$ any node in $S_i$
    end
    $\mathsf{IPDOM} \leftarrow \mathsf{ipdom}(G_S)$ by any efficient postdominance algorithm
      for graphs with unique end-node
    for $m \in N \setminus S$ do
        $\mathsf{ISINKDOM}[n] \leftarrow \mathsf{IPDOM}[n]$
    end
    for $1 \le i \le n$ do
        $n_1, \ldots, n_k \leftarrow$ any ordering of $S_i$
        for $1 \le j \le k$ do
            $\mathsf{ISINKDOM}[n_j] \leftarrow n_{j+1 \mod k}$
        end
    end
    return ISINKDOM
end
```

Algorithm 7: An algorithm for the computation of some $<_{\text{SINK}}$.

Observation 5.4.2. Let G be any CFG. Then Algorithm 7 terminates with a result $<_{\text{SINK}}$ s.t. $>_{\text{SINK}}$ is a transitive reduction of $\sqsupseteq_{\text{SINK}}$.

Remark 5.4.1. Wagner originally proposed to use the construction in Observation 5.4.1 to compute $\rightarrow_{\text{nticd}}$ directly, using any $\sqsupseteq_{\text{POST}}$-control dependence algorithm on G_S. This is possible, as well.

Summary

- Nontermination sensitive and insensitive postdominance $\sqsupseteq_{\text{MAX}}$ and $\sqsupseteq_{\text{SINK}}$ can be defined uniformly, as the least and greatest fixed point of a single functional.
- $\sqsupseteq_{\text{MAX}}$ and $\sqsupseteq_{\text{SINK}}$ transitively reduce not to trees, but to pseudoforests.
- These can be used to obtain least- and greatest fixed point algorithms.

6 Order Dependence

When you want to know how things really work, study them when they're coming apart.

(William Gibson — Zero History)

In the context of this thesis, the ultimate purpose of control- and other dependencies is to statically establish information flow security. In fact it is well known that — given a CFG with unique exit node n_x — non-termination insensitive, batch-style *non-interference* is soundly approximated by *slicing* w.r.t standard control dependence $\rightarrow_{\text{cd}}$ and data dependence $\rightarrow_{\text{data}}$ (see, e.g., [Was10]). Informally, in this setting

$$\text{IFC} = (\rightarrow_{\text{cd}} \cup \rightarrow_{\text{data}})^*$$

Nontermination sensitive control dependence $\rightarrow_{\text{ntscd}}$ and nontermination sensitive control dependence $\rightarrow_{\text{ntscd}}$ were explicitly designed for CFG without unique exit node n_x, in the extended setting of interactive, non-terminating programs. Ranganath et al. found that here, it is *not* enough to move from standard control dependence $\rightarrow_{\text{cd}}$ to $\rightarrow_{\text{ntscd}}$ or $\rightarrow_{\text{nticd}}$. In this setting, the attacker does not only see the values of some program configurations at the termination of the program (i.e.: when control flow reaches n_x), but also observes events during execution of possibly never-terminating sections in the CFG (e.g., during the execution of control *sinks*). Specifically, the attacker observes the *order* of execution of some such nodes.

With regard to non-termination sensitive notions of IFC, in [Ran+07] the authors define the ternary notion of ***d**ecisive **o**rder **d**ependence*, written $n \rightarrow_{\text{dod}} (m_1, m_2)$, which formalizes the notion that n controls the *order* in which two nodes m_1, m_2 are executed. Backward-Slicing w.r.t $\rightarrow_{\text{dod}}$ then proceeds along the rule

$$\frac{m_1 \in S \qquad m_2 \in S \qquad n \rightarrow_{\text{dod}} (m_1, m_2)}{n \in S}$$

and in the nontermination sensitive setting, the authors show that (informally):

$$\text{IFC} = \text{BISIM} = (\to_{\text{ntscd}} \cup \to_{\text{dod}} \cup \to_{\text{data}})^*$$

by proving that any *sliced* program (w.r.t $\to_{\text{ntscd}}$, $\to_{\text{dod}}$, $\to_{\text{data}}$) is weakly *bi*-similar to it's unsliced original program.

As for the nontermination insensitive setting, in [Amt08] the authors define a ternary notion of ***w**eak **o**rder **d**ependence*, written $n \to_{\text{wod}} (m_1, m_2)$, and then establish in this setting:

$$\text{SIM} = (\qquad\qquad \to_{\text{wod}} \cup \to_{\text{data}})^*$$

i.e.: they show that the sliced program (w.r.t $\to_{\text{wod}}$, $\to_{\text{data}}$) can weakly simulate the unsliced original program (but *not* the other way around, because the original program may get stuck in an infinite loop without observable actions, while the sliced program can proceed because the loop was sliced away).

In Section 6.1, I review $\to_{\text{dod}}$, and then observe how those regions of a graph in which $\to_{\text{dod}}$ is non-empty can be characterized in terms of nontermination-sensitive postdominance $\geq_{\text{MAX}}$. This leads to practical algorithm for the computation of $\to_{\text{dod}}$.

Then in Section 6.2, I propose the new ***n**ontermination **s**ensitive **o**rder **d**ependence* relation $\to_{\text{ntsod}}$ which can substitute for $\to_{\text{dod}}$ during slicing, but is often smaller and more efficient to compute than $\to_{\text{dod}}$.

In Section 6.3, I quickly review the bisimulation soundness criterion for nontermination sensitive slicing in *labeled* control flow graphs from [Ran+07], and then propose a new *trace* based soundness criterion for nontermination sensitive slicing in *unlabeled* graphs. With regard to this criterion, the slices obtained from $\to_{\text{ntsod}}$ and $\to_{\text{dod}}$ (together with $\to_{\text{ntscd}}$) will not only be sound but also *minimal*.

Later in Section 6.4, I will argue that $\to$wod is not appropriate for the purpose of information flow control, both for principal and for practical reasons. In particular:

- There exist always-terminating programs for which the $\to$wod, $\to$data backward slice of publicly "observable" nodes contain no private nodes, yet still, intuitively, private information is leaked.
- The relation $\to$wod is very large even for CFG with unique exit node n_x.

Because it is difficult to formulate a trace-based soundness criterion for nontermination insensitive slicing, I first suggest in Section 6.5 a simple criterion based on the notion of *next observable* nodes.

Only then in Section 6.6 I offer my attempt to actually define a trace based criterion for nontermination *in*sensitive slicing, using a notion of *infinite delay*.

In Section 6.7 I define ***n**ontermination **i**nsensitive **o**rder **d**ependence* ($\to$ntiod) such that in the non-termination insensitive setting

$$\text{IFC} = \text{INFDEL} = \text{NEXTOBS} = \left(\to\text{nticd} \;\cup\; \to\text{ntiod} \;\cup\; \to\text{data}\right)^*$$

The relation $\to$ntiod will in practice be much smaller than $\to$wod. In particular, and unlike $\to$wod, the relation $\to$ntiod is empty for CFG with unique exit node n_x. The relation $\to$ntiod may still become large for CFG with large control-sinks, but at least I will provide an algorithm for the computation of $\to$ntiod that in practice scales no worse than this size. This algorithm will be based on $\text{SINK}_{\text{down}}$ of Algorithm 6 from Section 5.2.

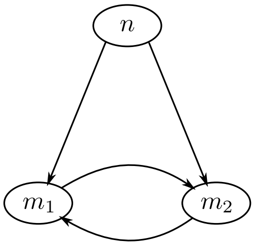


Figure 6.1: The canonical irreducible graph, where neither $n \rightarrow_{\text{ntscd}} m_1$ nor $n \rightarrow_{\text{ntscd}} m_2$.

6.1 Decisive Order Dependence

In [Ran+07], the authors note that for graphs such the canonical irreducible graph (Figure 6.1), nontermination sensitive control dependence $\rightarrow_{\text{ntscd}}$ is not enough to guarantee observational equivalence: there, neither $n \rightarrow_{\text{ntscd}} m_1$ nor $n \rightarrow_{\text{ntscd}} m_2$, but — assuming that both m_1 and m_2 are observable — the decision made at n will decide whether the sequence $m_1, m_2, m_1, \ldots$ or the sequence $m_2, m_1, m_2, \ldots$ is observed. In other words: An observer making either of these observations will learn what decision was made at n.

This is not merely a technical problem with $\rightarrow_{\text{ntscd}}$. In fact, *no* binary (dependence) relation can precisely capture observable equivalence for such graphs (also see: Observation 6.7.1 on page 99, or section 3.2 in [Ran+07]).

The authors remedy this by introducing the notion of ***d**ecisive **o**rder **d**ependence*, a ternary relation $\rightarrow_{\text{dod}}$ where $n \rightarrow_{\text{dod}} (m_1, m_2)$ iff n decisively decides in which order m_1 and m_2 may be observed. In order to state their formal definition, I need the following notation:

Definition 6.1.1. Given a path π, I write $m_1 \sqsupseteq_{m_2} \pi$ if π contains m_1 before any occurrence of m_2, i.e.:

- $m_1 \in \pi$
- for the shortest prefix π_0 of π such that $m_1 \in \pi_0$: $m_2 \notin \pi_0$

Definition 6.1.2 ($\rightarrow_{\text{dod}}$, ([Ran+07])). Let G be any CFG, and n, m_1, m_2 be distinct nodes. Then m_1, m_2 are decisively order dependent on n, written: $n \rightarrow_{\text{dod}} (m_1, m_2)$, iff

(a) All maximal paths from n contain both m_1 and m_2, i.e.:

$$m_1 \sqsupseteq_{\text{MAX}} n \quad \text{and} \quad m_2 \sqsupseteq_{\text{MAX}} n$$

(b) There exists some successor n_l of n such that all maximal paths π_l starting in n_l contain m_1 before any occurrence of m_2, i.e.:

$$\exists n \rightarrow_G n_l.\ \forall \pi \in {}_{n_l}\Pi_{\text{MAX}}.\ m_1 \sqsupseteq_{m_2} \pi$$

(c) There exists some successor n_r of n such that all maximal paths π_r starting in n_r contain m_2 before any occurrence of m_1, i.e.:

$$\exists n \rightarrow_G n_r.\ \forall \pi \in {}_{n_r}\Pi_{\text{MAX}}.\ m_2 \sqsupseteq_{m_1} \pi$$

The authors proof that $\rightarrow_{\text{dod}}$ is no larger than it needs to be, in the following sense:

Lemma 6.1.1 (([Ran+07]), Lemma 3)**.** Let G be reducible. Then $\rightarrow_{\text{dod}}$ is empty.

Unfortunately, this does not characterize the "regions" where triples $n \rightarrow_{\text{dod}} (m_1, m_2)$ may be found if G is *not* reducible. But such a characterization, presumably, is necessary for any efficient algorithm computing $\rightarrow_{\text{dod}}$. I now give such a characterization.

Lemma 6.1.2. Let G be any CFG.

(i) Whenever $n \rightarrow_{\text{dod}} (m_1, m_2)$, then

$$m_1 \sqsupseteq_{\text{MAX}} m_2 \quad \text{and} \quad m_2 \sqsupseteq_{\text{MAX}} m_1$$

(ii) Whenever $n \rightarrow_{\text{dod}} (m_1, m_2)$, and $m \sqsupseteq_{\text{MAX}} n$ for $m \neq n$, then

$$m_1 \sqsupseteq_{\text{MAX}} m \quad \text{and} \quad m \sqsupseteq_{\text{MAX}} m_1$$

as well as

$$m_2 \sqsupseteq_{\text{MAX}} m \quad \text{and} \quad m \sqsupseteq_{\text{MAX}} m_2$$

(iii) Whenever $n \rightarrow_{\text{dod}} (m_1, m_2)$, then

$$\text{neither} \quad n \sqsupseteq_{\text{MAX}} m_1 \quad \text{nor} \quad n \sqsupseteq_{\text{MAX}} m_2$$

Proof: On page 358 in the appendix. □

As shown before in Section 5.2, $\sqsupseteq_{\text{MAX}}$ can be efficiently computed and represented as transitive reduction $>_{\text{MAX}}$, which turned out to be a pseudo-forest. The situation in Lemma 6.1.2, rephrased in terms of $>_{\text{MAX}}$ — is depicted in Figure 6.2, and formally reads:

Corollary 6.1.1. Let G be any CFG, and $>_{\text{MAX}}$ any transitive reduction of $\sqsupseteq_{\text{MAX}}$.

For $n \rightarrow_{\text{dod}} (m_1, m_2)$, let $M = \{ m \mid m_1 <^*_{\text{MAX}} m \} = \{ m \mid m_2 <^*_{\text{MAX}} m \}$ be the $<_{\text{MAX}}$-cycle both m_1, m_2 are part of. Then

- $n \notin M$
- $n <_{\text{MAX}} m$ for some $m \in M$.

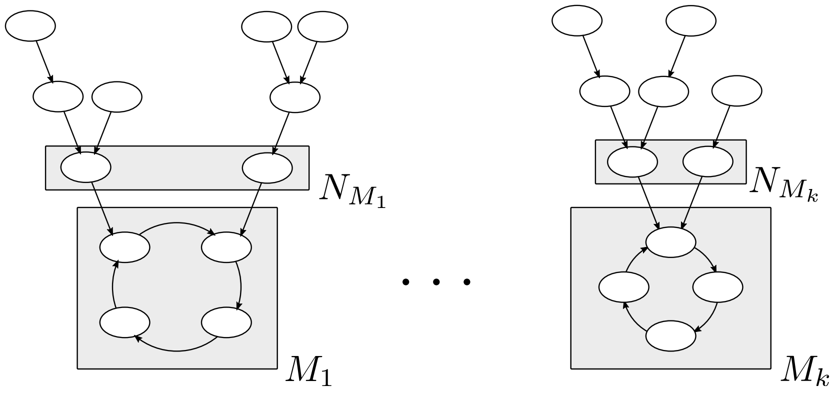


Figure 6.2: The situation in Corollary 6.1.1, exemplified. Given $m_1, m_2 \in M_i$, the set of n such that $n \rightarrow_{\text{dod}} (m_1, m_2)$ is contained in N_{M_i}.

6.1.1 Algorithms

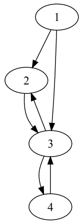
In [Ran+07], the authors propose a semi-naive algorithm (Figure 7, page 38) for the computation of $\rightarrow$dod. As a subroutine, it implements a check DEP (n, m_1, m_2)[1] such that

$$\text{DEP}\,(n, m_1, m_2) \iff \text{clauses (b), (c) in Definition 6.1.2 hold}$$

The author then propose to compute[2] $\rightarrow$dod as the set of all triples (n, m_1, m_2) such that

$$n \in \text{COND},\ m_1 \rightarrow_G^* m_2,\ m_2 \neq m_1,\ m_2 \rightarrow_G^* m_1,\ \text{DEP}\,(n, m_1, m_2) \quad (6.1)$$

Let me first point out that this is incorrect in general. An example is shown on the right. Here, $\neg\ 3 \rightarrow_{\text{dod}} (2, 4)$, since while clauses (b) and (c) hold, clause (a) does *not*: Neither $2 \sqsupseteq_{\text{MAX}} 3$ nor $3 \sqsupseteq_{\text{MAX}} 2$.



[1] originally: DEPENDENCE
[2] via "generate-and-test" in the implied order

The obvious fix is to replace the checks

$$m_1 \to_G^* m_2,\ m_2 \to_G^* m_1 \quad \text{by} \quad m_1 \sqsupseteq_{\text{MAX}} n,\ m_2 \sqsupseteq_{\text{MAX}} n$$

and obtain

$$n \in \text{COND},\ m_1, m_2 \sqsupseteq_{\text{MAX}} n,\ m_2 \neq m_1,\ \mathsf{DEP}\,(n, m_1, m_2) \tag{6.2}$$

but the authors do not provide an algorithm to check (let alone: enumerate such m_i given n) this. Given my Algorithm 5 for the computation of some $<_{\text{MAX}}$, however, this is both trivial.

Furthermore, Corollary 6.1.1 allows me to significantly reduce the number of queries $\mathsf{DEP}\,(n, m_1, m_2)$. Let $>$ be some transitive reduction of $\sqsupseteq_{\text{MAX}}$, and $\mathbb{M}$ be the set of $<$-cycles M, which are easily enumerated given $<$. Also easily enumerated for each M is the set $N_M = \{\, n \in \text{COND} \mid n \notin M,\ \exists m \in M.\ n < m \,\}$. The resulting scheme — which can be further optimized by noting that $\to$dod is symmetric w.r.t m_1, m_2 — then is:

$$M \in \mathbb{M},\ m_1 \neq m_2 \in M,\ n \in N_M,\ \mathsf{DEP}\,(n, m_1, m_2) \tag{6.3}$$

Using my own implementation of the algorithm DEP (which I will not repeat, here), the empirical comparison of (6.2) and (6.3) confirms Lemma 6.1.2:

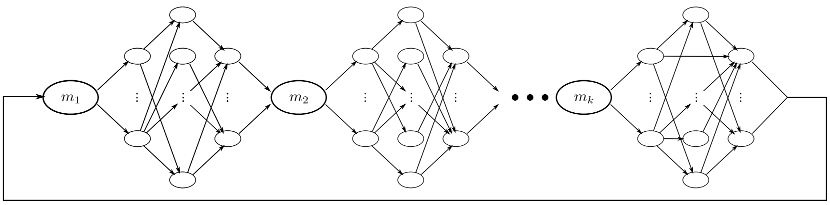
Observation 6.1.1. Given any CFG, the relation computed via 6.2 equals that computed via 6.3.

Given a CFG $G = (N, E)$, the complexity of DEP is, in [Ran+07], given as $\mathcal{O}\left(|E| \times |N| \times \log\left(|N|\right)\right)$.

The original (flawed) scheme (6.1) requires $\Theta\left(|N|^3\right)$ queries to DEP, while for my scheme (6.3), the number of queries is highest if the maximal size

$$\max_{M \in \mathbb{M}} |M|$$

among $<_{\text{MAX}}$-cycles M is maximized (since each M and N_M are disjunct). Hence, the scheme (6.3) requires $\mathcal{O}\left(|N|^2\right)$ queries to DEP, but I claim that in CFG for virtually any programming language and any but the most untypical program, $\max_{M \in \mathbb{M}} |M|$ is much smaller than $|N|$. This is because any $<_{\text{MAX}}$-cycle $M = \{m_1, \ldots, m_k\}$ is just the spine of a sub-cycle free control-sink, i.e., a G-subgraph of the form



6.2 Nontermination Sensitive Order Dependence

I do not attempt to further improve upon the computation scheme (6.3) for $\rightarrow_{\text{dod}}$ I gave in Subsection 6.1.1.

Instead, in this section, I propose a new (indecisively) **n**on**t**ermination **s**ensitive **o**rder **d**ependence relation $\rightarrow_{\text{ntsod}}$ which, in the application of information flow control, can be substituted for $\rightarrow_{\text{dod}}$. Informally:

$$(\rightarrow_{\text{ntscd}} \cup \rightarrow_{\text{dod}})^* \quad = \quad (\rightarrow_{\text{ntscd}} \cup \rightarrow_{\text{ntsod}})^*$$

For this new relation $\rightarrow_{\text{ntsod}}$, I will give an efficient-in-practice algorithm. This will be possible because I can rephrase $\rightarrow_{\text{ntsod}}$ in terms of $\rightarrow_{\text{ntscd}}$ for certain subgraphs of G. Hence, I can once again use Algorithm 1 for the computation of suitable generalized postdominance frontiers, based on Algorithm 5 (or Algorithm 19).

Definition 6.2.1 ($\rightarrow_{\text{ntsod}}$)**.** Let G be any CFG, and n, m_1, m_2 be distinct nodes. Then m_1 is **n**on**t**ermination **s**ensitively **o**rder **d**ependent on n with respect to m_2, written: $n \rightarrow_{\text{ntsod}} (m_1, m_2)$, iff

(a) All maximal paths from n contain both m_1 and m_2, i.e.:

$$m_1 \sqsupseteq_{\text{MAX}} n \quad \text{and} \quad m_2 \sqsupseteq_{\text{MAX}} n$$

(b) There exists some successor n_l of n such that all maximal paths π_l starting in n_l contain m_1 before any occurrence of m_2, i.e.:

$$\exists n \rightarrow_G n_l. \quad \forall \pi \in {}_{n_l}\Pi_{\text{MAX}}.\ m_1 \sqsupseteq_{m_2} \pi$$

(c) There exists some successor n_r of n such that *not* all maximal paths π_r starting in n_r contain m_1 before any occurrence of m_2, i.e.:

$$\exists n \rightarrow_G n_r.\ \neg\ \forall \pi \in {}_{n_r}\Pi_{\text{MAX}}.\ m_1 \sqsupseteq_{m_2} \pi$$

Note that (as opposed to $\rightarrow_{\text{dod}}$) this definition is *not* symmetric in m_1, m_2. Also note that after holding fast m_2, and subject to the constraint (a), the binary relation $\cdot \rightarrow_{\text{ntsod}} (\cdot, m_2)$ follows the scheme of $\sqsupseteq$ control dependence (established in Definition 3.1.2), using $\sqsupseteq = \sqsupseteq_{\text{MAX}[m_2]}$ and the following notation:

Definition 6.2.2 ($m_1 \sqsupseteq_{\text{MAX}[m_2]} n$)**.** Let G be some CFG.

$$m_1 \sqsupseteq^{G}_{\text{MAX}[m_2]} n \quad \Leftrightarrow \quad \forall \pi \in {}_{n}\Pi_{\text{MAX}}.\ m_1 \sqsupseteq_{m_2} \pi$$

A minor modification to the rule system D from Theorem 5.1.2 yields the following rule system:

Proposition 6.2.1. Let G be a CFG, m_2 any of it's nodes, and D_{m_2} be the rule-system

$$\frac{n \neq m_2}{n \sqsupseteq n}\,\mathsf{D}^{\text{self}}_{m_2} \qquad \frac{\forall p \rightarrow_G x.\ m \sqsupseteq x \qquad p \rightarrow^{*}_{G} m \qquad p \neq m_2}{m \sqsupseteq p}\,\mathsf{D}^{\text{suc}}_{m_2}$$

Then $\sqsupseteq_{\text{MAX}[m_2]} = \mu \mathsf{D}_{m_2}$.

6.2.1 Comparison with Decisive Order Dependence

The relation $\rightarrow_{\text{ntsod}}$ is meant to replace $\rightarrow_{\text{dod}}$. In fact, its symmetric core is just $\rightarrow_{\text{dod}}$:

Observation 6.2.1. Let G be any CFG. Then

$$n \rightarrow_{\text{dod}} (m_1, m_2) \iff n \rightarrow_{\text{ntsod}} (m_1, m_2) \ \wedge\ n \rightarrow_{\text{ntsod}} (m_2, m_1)$$

At the same time, $\rightarrow_{\text{ntsod}}$ is not too large for application to information flow control:

Observation 6.2.2. Let G be any CFG, and assume $n \rightarrow_{\text{ntsod}} (m_1, m_2)$, but $\neg\ n \rightarrow_{\text{ntsod}} (m_2, m_1)$. Then there exists some n' with both $n' \rightarrow_{\text{ntsod}} (m_1, m_2)$ and $n' \rightarrow_{\text{ntsod}} (m_2, m_1)$ such that

$$n \rightarrow^{*}_{\text{ntscd}} n'$$

This observation immediately implies that slicing backwards from any set M along $\rightarrow$nticd, $\rightarrow$dod is equivalent to slicing backwards along $\rightarrow$nticd, $\rightarrow$ntsod. I use the following notation to state this as Proposition 6.2.2:

Definition 6.2.3. Given a binary relation $\cdot \rightarrow \cdot$ and a ternary relation $\cdot \Rightarrow (\cdot, \cdot)$ on nodes N, their backward slice — written $(\rightarrow\ \dot\cup\ \Rightarrow)^*$ — is the function mapping any set $M \subseteq N$ to the smallest set $S \supseteq M$ satisfying

$$\frac{n \rightarrow m \qquad m \in S}{n \in S} \qquad \frac{n \Rightarrow (m_1, m_2) \qquad m_1 \in S \qquad m_2 \in S}{n \in S}$$

I write both

$$\begin{array}{ccc} & (\rightarrow)^* & \mathit{for} & (\rightarrow\ \dot\cup\ \emptyset)^* \\ \mathit{and} & (\Rightarrow)^* & \mathit{for} & (\emptyset\ \dot\cup\ \Rightarrow)^* \end{array}$$

The following result directly follows from Observation 6.2.1 and Observation 6.2.2.

Proposition 6.2.2.

$$(\rightarrow\text{ntscd}\ \dot\cup\ \rightarrow\text{dod})^* = (\rightarrow\text{ntscd}\ \dot\cup\ \rightarrow\text{ntsod})^*$$

Structural Properties

Structurally, the relation $\rightarrow$ntsod is constrained to the very same regions within $>_{\text{MAX}}$ that $\rightarrow$dod is:

Observation 6.2.3. Lemma 6.1.2 still holds after replacing all occurrences of $n \rightarrow_{\text{dod}} (m_1, m_2)$ with $n \rightarrow_{\text{ntsod}} (m_1, m_2)$.

6.2.2 An Algorithm Based on Nontermination Sensitive Postdominance

Owing to Observation 6.2.3, I can limit the search for triples $n \rightarrow_{\text{ntsod}} (m_1, m_2)$ the same way I previously (in Subsection 6.1.1) did for $\rightarrow_{\text{dod}}$. But I can do more: for fixed m_2, the relation $\cdot \rightarrow_{\text{ntsod}} (\cdot, m_2)$ is essentially $\sqsupseteq_{\text{MAX}[m_2]}$-control dependence, with $\sqsupseteq_{\text{MAX}[m_2]}$ *almost* equal to $\sqsupseteq_{\text{MAX}}$:

$$\begin{array}{rlcl} & \forall \pi \in {}_{n_s}\Pi_{\text{MAX}}.\ m_1 \sqsupseteq_{m_2} \pi & \Leftrightarrow & m_1 \sqsupseteq_{\text{MAX}[m_2]} n_s \\ \text{instead of} & \forall \pi \in {}_{n_s}\Pi_{\text{MAX}}.\ m_1 \in \pi & \Leftrightarrow & m_1 \sqsupseteq_{\text{MAX}} n_s \end{array}$$

By choosing — given m_2 — a suitable subgraph of G, it is not hard to reduce $\sqsupseteq_{\text{MAX}[m_2]}$ in G to $\sqsupseteq_{\text{MAX}}$ in the subgraph. To do this, I use the following notation:

Definition 6.2.4. Let $G = (N, E)$ be some CFG, $n, m \in N$, $M \subseteq N$. Then I define

$$\begin{aligned} G_{M\not\rightarrow} &= (N, E \setminus \{ (m, n) \mid n \in N,\ m \in M \}) \\ G_{m\not\rightarrow} &= G_{\{m\}\not\rightarrow} \end{aligned}$$

In other words: $G_{M\not\rightarrow}$ is the subgraph obtained from G by deleting all outgoing edges of nodes in M.

$$\begin{aligned} N^{\rightarrow^* M} &= \{ n \mid \exists m \in M.\ n \rightarrow^*_G m \} & N^{M \rightarrow^*} &= \{ n \mid \exists m \in M.\ m \rightarrow^*_G n \} \\ G^{\rightarrow^* M} &= \left(N^{\rightarrow^* M}, E|_{N^{\rightarrow^* M}} \right) & G^{M \rightarrow^*} &= \left(N^{M \rightarrow^*}, E|_{N^{M \rightarrow^*}} \right) \end{aligned}$$

$$\begin{aligned} N^{\rightarrow^* m} &= N^{\rightarrow^* \{m\}} & N^{m \rightarrow^*} &= N^{\{m\} \rightarrow^*} \\ G^{\rightarrow^* m} &= G^{\rightarrow^* \{m\}} & G^{m \rightarrow^*} &= G^{\{m\} \rightarrow^*} \end{aligned}$$

$G^{\to^* M}$ is the subgraph of G consisting of those nodes that can reach some node in M, and $G^{M\to^*}$ is the subgraph of G consisting of those nodes that area reachable from some node in M.

$$\begin{aligned} G^{\to^* M}_{m\not\to} &= \left(G^{\to^* M}\right)_{m\not\to} & G^{M_1\to^* M_2} &= \left(G^{M_1\to^*}\right)^{\to^* M_2} \\ G^{M\to^*}_{m\not\to} &= \left(G^{M\to^*}\right)_{m\not\to} & G^{\to^* M}_{M'\not\to} &= \left(G^{\to^* M}\right)_{M'\not\to} \end{aligned}$$

$G^{M_1\to^* M_2}$ is the "chop" subgraph of G between M_1 and M_2, and $G^{M\to^*}_{m\not\to}$ is the graph obtained from G by retaining only those nodes that are reachable from some node in M, and deleting all outgoing edges of node m.

Observation 6.2.4. Let $G = (N, E)$ be any CFG, and $n_s, m_1, m_2 \in N$, $m_1 \neq m_2$. Then any maximal G-path starting in n_s contains m_1 before any occurrence of m_2 iff *in the graph $G_{m_2\not\to}$ obtained by removing all outgoing edges of m_2*, any maximal path starting in n_s contains m_1, i.e.:

$$m_1 \sqsupseteq^{G}_{\mathrm{MAX}[m_2]} n_s \quad\iff\quad m_1 \sqsupseteq^{G_{m_2\not\to}}_{\mathrm{MAX}} n_s$$

Combining Observation 6.2.3 and Observation 6.2.4 in a form suggesting an algorithm, I make the following observation:

Observation 6.2.5. Let $G = (N, E)$ be any CFG, $> \; = \; >_{\mathrm{MAX}}$ any transitive reduction of $\sqsupseteq^{G}_{\mathrm{MAX}}$, $\mathbb{M}$ the set of $<$-cycles, and let — as before in Figure 6.2 — for $M \in \mathbb{M}$ the set of conditional nodes $n \notin M$ s.t. $n < m$ for some $m \in M$ be denoted by N_M.

Then, $n \to_{\mathrm{ntsod}} (m_1, m_2)$ if and only if in the CFG

$$G_{m_2} := G^{N_M\to^* M}_{m_2\not\to}$$

m_1 is $\to_{\mathrm{ntscd}}$-control dependent on n, where $M \in \mathbb{M}$ is the $<$-cycle containing both m_1 and m_2, i.e.:

$$n \to_{\mathrm{ntsod}} (m_1, m_2) \;\Leftrightarrow\; \exists M \in \mathbb{M}.\, m_1, m_2 \in M \,\wedge\, n \in N_M \,\wedge\, n \to^{G_{m_2}}_{\mathrm{ntscd}} m_1$$

Informally:

$$\rightarrow\text{ntsod} \quad \approx \quad \left(\sum_{M \in \mathbb{M}} |M|\right) \times \rightarrow\text{ntscd}$$

6.3 Soundness of Nontermination Sensitive Slices

In [Ran+07], the soundness criterion for nontermination sensitive slicing requires a weak bisimulation between the original and sliced program. The authors label CFG-nodes with statements reading and modifying *states* $\sigma \in \Sigma$, and assume a corresponding CFG-execution semantics $G \vdash s \rightarrow s'$ for $s, s' \in N \times \Sigma$. Given a set S of (observable) nodes, they differentiate between S-visible, and silent (τ) transitions:

Definition 6.3.1 ([Ran+07], Definition 20)**.**

- for $s = (n, \sigma)$, $G \vdash s \overset{n}{\longmapsto} s'$ iff $G \vdash s \rightarrow s' \wedge n \in S$
- for $s = (n, \sigma)$, $G \vdash s \overset{\tau}{\longmapsto} s'$ iff $G \vdash s \rightarrow s' \wedge n \notin S$
- $G \vdash \overset{\tau}{\Longmapsto} \quad = \quad G \vdash \overset{\tau}{\longmapsto}^{*}$
- $G \vdash s \overset{n}{\Longmapsto} s''$ iff $G \vdash s \overset{\tau}{\Longmapsto} s'$, $G \vdash s' \overset{n}{\longmapsto} s''$ for some s'

Furthermore, given a set S of nodes and statement-labeled CFG G, the authors define the corresponding *sliced* (labeled) CFG G_S such that labels remain unchanged for $n \notin S$, and become *no-ops*[3] for $n \in S$.

Then, the authors show that, for S closed under $\rightarrow_{\text{ntscd}}$, $\rightarrow_{\text{data}}$, $\rightarrow_{\text{dod}}$, there exists some weak bisimulation R between G and G_S, i.e. a binary relation R on configurations s such that:

- If $s_1 \; R \; s_2$ and $G \vdash s_1 \overset{\tau}{\longmapsto} s_1'$, then $G_S \vdash s_2 \overset{\tau}{\Longmapsto} s_2'$ for some s_2' such that $s_1' \; R \; s_2'$, and
  if $s_1 \; R \; s_2$ and $G_S \vdash s_2 \overset{\tau}{\longmapsto} s_2'$, then $G \vdash s_1 \overset{\tau}{\Longmapsto} s_1'$ for some s_1' such that $s_1' \; R \; s_2'$
- If $s_1 \; R \; s_2$ and $G \vdash s_1 \overset{n}{\longmapsto} s_1'$, then $G_S \vdash s_2 \overset{n}{\Longmapsto} s_2'$ for some s_2' such that $s_1' \; R \; s_2'$, and

[3] i.e.: operations that leave the state component σ unchanged

if $s_1 \; R \; s_2$ and $G_S \vdash s_2 \overset{n}{\longmapsto} s_2'$, then $G \;\vdash s_1 \overset{n}{\Longmapsto} s_1'$ for some s_1' such that $s_1' \; R \; s_2'$

Thanks to Proposition 6.2.2, the same holds for S closed under $\rightarrow_{\text{ntscd}}$, $\rightarrow_{\text{data}}$, $\rightarrow_{\text{ntsod}}$, so I am almost ready to conclude my treatment of nontermination sensitive order dependencies.

Before I do that, however, I will observe that in the simplified setting of unlabeled CFG (i.e.: those without statements), slicing w.r.t $\rightarrow_{\text{ntscd}}$, $\rightarrow_{\text{ntsod}}$ (but not: $\rightarrow_{\text{data}}$, since data dependencies no longer make sense) satisfies a different *trace based* criterion.

The advantages of this criterion will be that

1. At least for very small CFG[4] ($|N| \leq 25$), it can be exhaustively tested.
2. It will turn out that — with regards to this criterion — slices will not only be sound, but also *minimal*. [5]

Definition 6.3.2. Let G be any (unlabeled) CFG, and $N_x = \{\, n_x \mid \neg\exists m. n \rightarrow_G m \,\}$ be the set of exit nodes. An *input* i to G is a pair $i = (n_e, \epsilon)$ of a node n_e (the entry node) and a map $\epsilon \;:\; \text{COND}_G \rightarrow N$ (the choice made by i at n) such that $n \rightarrow_G \epsilon(n)$ for all conditional nodes n.

Given input i, the deterministic transition relation

- $G, i \vdash n \rightarrow n'$ if $n' = \epsilon(n)$ for $n \in \text{COND}_G$
- $G, i \vdash n \rightarrow n'$ if $n \rightarrow_G n'$ for $n \notin \text{COND}_G$

[4] and sets of nodes S arising as backward slides from small slicing criteria $M \subseteq S$

[5] presumably, $\rightarrow_{\text{ntscd}}$, $\rightarrow_{\text{data}}$, $\rightarrow_{\text{ntsod}}$-slices are also minimal for statement-labeled CFG *up to the def-use abstraction*, i.e.: whenever $n \in S$, there exists *some* labeling for the same CFG with the same def- and use sets such that the newly-labeled CFG behaves observably differently for two observably-equivalent inputs. A corresponding result for syntax-tree based slicing is given [Dan+05], but I am unaware of any prior result with regard to the slicing of arbitrary CFG.

determines a unique (possibly infinite) path in G

$$\pi_i^G = n_e, \ldots$$

such that $n \in \textsc{cond}_G$ implies $n' = \epsilon(n)$ for any two consecutive nodes n, n'.

Similarly, it determines a unique sequence t_i^G of *partial edges* $(n, n') \in \ E \cup (N_x \times \{\bot\})$ that is either finite with — for some exit node n_x —

$$t_i^G = (n_e, n_1)\,,\ (n_1, n_2)\,,\ \ldots,\ (n_k, n_x)\,,\ (n_x, \bot)$$

or infinite with

$$t_i^G = (n_e, n_1)\,,\ (n_1, n_2)\,,\ \ldots$$

such that $n \in \textsc{cond}_G$ implies $n' = \epsilon(n)$ for any edge (n, n'). I call such sequences *traces*.

Remark 6.3.1. Given such an input i, the trace t_i in G is uniquely determined. This is not so different from models of labeled CFG with deterministic state transitions at non-conditional nodes, and mutually exclusive (and: usually exhaustive) choices (depending on state σ) made at conditional nodes. Usually in those models, the entry node n_e is fixed, and the input consists of an initial state σ_0.

Definition 6.3.3. Given an input $i = (n_e, \epsilon)$ to G and a set S of ("observable") nodes, the S-observable input $i\big|_S$ of i is the restriction $\left(n_e, \epsilon\big|_S\right)$ of the choices ϵ to nodes S. Furthermore, two inputs i, i' in G are called S-equivalent, and I write $i \sim_S i'$, iff

$$i\big|_S = i'\big|_S$$

Definition 6.3.4. Given a (possibly infinite) sequence t of partial edges, and a set S of (observable) nodes, the S-observable subsequence $t\big|_S$ of t is obtained from t by removing any occurrences of (partial) edges $(n,\_)$ s.t. $n \notin S$ from t (possibly transforming an infinite sequence into a finite).

Furthermore, two traces t, t' in G are called nontermination sensitively S-equivalent, and I write $t \sim_S t'$, iff

$$t\big|_S = t'\big|_S$$

Because in this "stateless" notion of input and execution of CFG, choices $\epsilon(n)$ made at $n \in \textsc{cond}$ are deterministic (depending only on n, and not on some state σ), any path π_i is either

- a finite path of the form

  $$\pi_{\mathsf{fin}} = n_e, \ldots, n_x$$

  for some n_x such that $\neg\, n_x \rightarrow_G$, or

- an infinite path of the form

  $$\pi_{\mathsf{inf}} = \underbrace{n_e, \ldots, n}_{\pi_{\mathsf{fin},0}}, \pi, \pi, \ldots$$

  with a finite prefix $\pi_{\mathsf{fin},0}$ and an infinitely-repeating, finite and minimal cycle-segment π, i.e.: a path

  $$\pi = n', \ldots, m$$

  with no node occurring twice, and such that $n \rightarrow_G n'$ and $m \rightarrow_G n'$

This fact allows me, given any input i, to finitely[6] represent its execution path π_i via $\gamma\left(\pi_{fin}\right) = \pi_{fin}$ and $\gamma\left(\pi_{fin,0}, \pi\right) = \pi_{inf}$. I can represent any execution *trace* t_i similarly.

[6] and: uniquely

It is self-evident that, given such a finite representation of some t, the finite representation of $t|_S$ can be efficiently computed. Also, as will be required later, it can be efficiently checked whether $t \sqsubseteq_{\mathcal{T}} t'$, i.e.: whether t is a prefix of t', and the finite representation of the concatenation $t\ t'$ can be computed efficiently, with $tt' = t$ if t is infinite.

In summary, I can confirm the following observation empirically:

Observation 6.3.1 (Soundness of $\to_{\text{ntsod}}, \to_{\text{ntscd}}$)**.** Let G be any CFG, and $M \subseteq N$ a set of nodes (the slicing criterion). Let $S = (\to_{\text{ntscd}} \cup\!\!\!\cdot\ \to_{\text{ntsod}})^* (M)$ be the backward slice w.r.t M. Then, for any inputs i, i' such that

$$i \sim_S i'$$

I have

$$t_i \sim_S t_{i'}$$

Observation 6.3.2 (Minimality of $\to_{\text{ntsod}}, \to_{\text{ntscd}}$)**.** Let G be any CFG, and $M \subseteq N$ a set of nodes (the slicing criterion). Let $S = (\to_{\text{ntscd}} \cup\!\!\!\cdot\ \to_{\text{ntsod}})^* (M)$ be the backward slice w.r.t M. Then, for any $n \in S, n \notin M$ and $S' = S \setminus \{n\}$, there exist inputs i, i' such that

$$i \sim_{S'} i'$$

but:

$$\neg\ t_i \sim_{S'} t_{i'}$$

The fact that $\to_{\text{ntscd}}$, $\to_{\text{ntsod}}$ slices are minimal also retrospectively establishes that in fact, *some* kind of ternary dependence relation is required, and there is not merely a problem with the binary relation $\to_{\text{ntscd}}$. To show why, I repeat in Figure 6.3 the canonical irreducible graph from Figure 6.1. Assume that there exists some binary relation $\to_{\text{xcd}}$ such that for all slicing criteria M, the slice $(\to_{\text{xcd}})^* (M)$ is sound and minimal. For $M = \{m_1, m_2\}$, I then have $(\to_{\text{xcd}})^* (M) = \{n, m_1, m_2\}$. Since $\to_{\text{xcd}}$ is binary, then necessarily $n \to_{\text{xcd}} m_1 \lor n \to_{\text{xcd}} m_2$. But if $n \to_{\text{xcd}} m_1$, then for the alternative

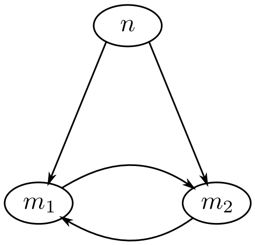


Figure 6.3: Neither $n \rightarrow_{\text{ntscd}} m_1$ nor $n \rightarrow_{\text{ntscd}} m_2$, but $n \rightarrow_{\text{ntsod}} (m_1, m_2)$ and $n \rightarrow_{\text{ntsod}} (m_2, m_1)$.

slicing criterion $M' = \{m_1\}$, I have $(\rightarrow_{\text{xcd}}\ )^* (M') \supseteq \{n, m_1\}$, which is *not* minimal, since $(\rightarrow\text{ntscd} \ \uplus \ \rightarrow\text{ntsod})^* (M') = \{m_1\}$.

Observation 6.3.3. Let $G = (N, E)$ be the canonical irreducible graph from Figure 6.1 on page 66. Then there exists no binary relation $\rightarrow$ such that for all $M \subseteq N$

$$(\rightarrow)^* (M) = (\rightarrow\text{ntscd} \ \uplus \ \rightarrow\text{ntsod})^* (M)$$

Remark 6.3.2. I must stress that when applying slicing (be it nontermination sensitive or not) for the purpose of information flow control, the set S of nodes deemed observable will usually *not* be implicitly defined to be the slice $S = (\ldots)^* (M)$ of some criterion M. Instead, the set of observable nodes will remain fixed and fully determined by a user-provided specification, and *not* depend on the chosen slicing technique. For example, the user might provide a set L of nodes deemed public (or: *low* observable), and a disjoint set H of secret (or: *high*-observable) nodes. Slicing will then determine whether

$$h \in (\ldots)^* (L)$$

for some $h \in H$, in which case I must expect some inputs i, i' with different choices $\epsilon (h) \neq \epsilon' (h)$ made at h made such that

$$i \sim_L i'$$

but:

$$\neg\, t_i \sim_L t_{i'}$$

If, on the other hand, no such h exists, i will conclude that any (L-observable) difference between $t_i\big|_L$ and $t_{i'}\big|_L$ is purely due to different choices $\epsilon\,(x) \neq \epsilon'\,(x)$ made at nodes $x \notin H$ (i.e.: that are *not* considered *high*), and hence that any L-observer cannot learn choices made at $h \in H$.

6.4 Weak Order Dependence

In this and the following sections, I turn from nontermination sensitive notions of order dependency to nontermination *in*sensitive notions. The first such notion for arbitrary CFG was proposed in [Amt08]:

Definition 6.4.1 ($\rightarrow_{\text{wod}}$)**.** Let G be any CFG, and n, m_1, m_2 be nodes. Then m_1, m_2 are **w**eakly **o**rder **d**ependent on n, written: $n \rightarrow_{\text{wod}} (m_1, m_2)$, iff

(a) There exists some path

$$\pi_1 = n, \ldots, m_1 \quad \text{with} \quad m_2 \notin \pi_1$$

(b) There exists some path

$$\pi_2 = n, \ldots, m_2 \quad \text{with} \quad m_1 \notin \pi_2$$

(c) There exists some successor n' of n such that one of the following holds:

- m_1 is reachable from n', and $m_1 \in \pi'_1$ for all paths

$$\pi'_1 = n', \ldots, m_2$$

- m_2 is reachable from n', and $m_2 \in \pi'_2$ for all paths

$$\pi'_2 = n', \ldots, m_1$$

Note that $n \rightarrow_{\text{wod}} (m_1, m_2)$ implies that n, m_1, m_2 are distinct. Also note that — unlike $\rightarrow_{\text{dod}}$ and $\rightarrow_{\text{ntsod}}$ —, nodes such that $n \rightarrow_{\text{wod}} (m_1, m_2)$ are *not* constraint to any particular regions of the CFG (e.g., $n \rightarrow_{\text{wod}} (m_1, m_2)$ does *not*, in general, imply that $m_1 \sqsupseteq_{\text{SINK}} m_2$ and $m_2 \sqsupseteq_{\text{SINK}} m_1$, or similar). In fact, $\rightarrow_{\text{wod}}$ is designed to "cover" the whole CFG, in the sense that slicing is meant *not* to additionally require closure w.r.t $\rightarrow_{\text{nticd}}$ (unlike slicing with $\rightarrow_{\text{dod}}$, which additionally required closure under $\rightarrow_{\text{ntscd}}$).

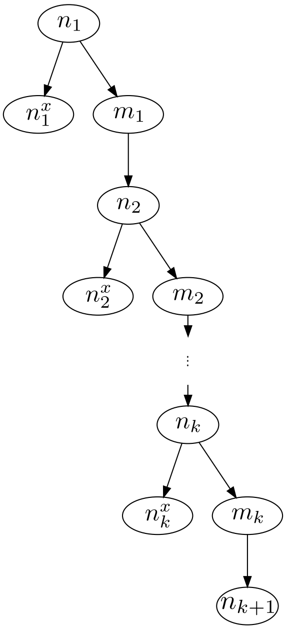
Assuming a statement-labeled CFG (see: Definition 6.3.1 etc. pp.), the authors proceed to show that, for S closed under $\rightarrow_{\text{data}}$, $\rightarrow_{\text{wod}}$, there exists some weak simulation R between G and G_S (i.e.: the labeled graph obtained from G by replacing statements of nodes $n \notin S$ with no-ops):

- If $s_1 \; R \; s_2$ and $G \;\vdash s_1 \overset{n}{\Longmapsto} s_1'$, then $G_S \vdash s_2 \overset{n}{\Longmapsto} s_2'$ for some s_2' such that $s_1' \; R \; s_2'$.

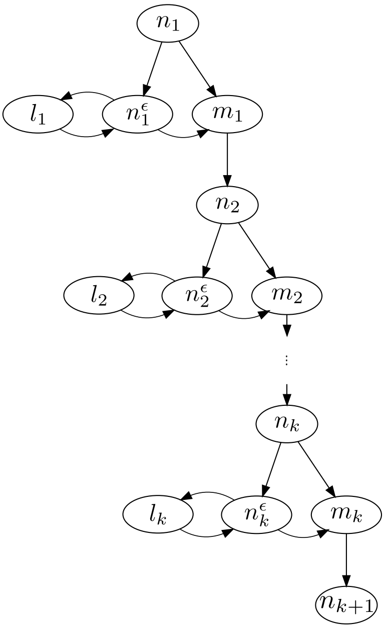
The key difference between this result and the corresponding result for nontermination sensitive slicing is that there, the original and the sliced graph were (weakly) *bi*-similar, i.e.: any observable step in the original graph could be matched in the sliced graph *and vice versa*, while weakly similarity for nontermination insensitive slicing as just stated does *not* demand observable steps in the sliced graph to be matched in the original graph, because non-termination may have been “sliced away”.

While this simulation-based correctness-notion is appropriate for *batch-style* non-interference for CFG with unique exit node n_x (see, e.g., [Was10] for formal proof), and while some notions of non-interference that allow observations before a programs termination are similar, (e.g.: TINI in [Ask+08]), I find it unsatisfactory because under this notion, even always-terminating programs may leak (even: “more than a bit”). To see this, consider Figure 6.4a, and imagine this to be the CFG of the program that at each conditional node n_i tests `h` $== i$ for some secret input variable `h`. Also, let $M = \{m_1, \ldots, m_k\}$ be the set of observable node. Then the observable trace of this CFG (assuming entry node $n_e = n_1$) is

$$(m_1, n_2), \ldots, (m_i, n_{i+1}) \text{ iff the input to } \texttt{h} \text{ is } i,$$



(a) The CFG of an always-terminating program



(b) The CFG of a possibly non-terminating program

Figure 6.4: The CFG of a program that leaks more than a bit

but it is easy to see that the simulation criterion holds for $S = M$. In fact, for any pair $m_i, m_{i'} \in M$, there exist *no* $n \in N$ such that $n \to_{\text{wod}} (m_i, m_{i'})$, so the set

$$S = M = (\to_{\text{data}} \cup \to_{\text{wod}})^* (M)$$

is closed under $\to_{\text{data}}$, $\to_{\text{wod}}$.

Remark 6.4.1. The example in Figure 6.4a can be understood to paraphrase the example *Program 2* from [Ask+08], show in Figure 6.5. I essentially unrolled the `for` loop, and replaced the `while` loop with a

```
for i = 0 to k (
  output i on public_channel
  if (i = secret) then (while true do skip)
)
```

Figure 6.5: Example from [Ask+08]

transition to an exit node n_i^x. In the light of this, I concede that my misgivings of the simulation criterion are possibly not fundamental, since it appears that any always-terminating, leaking program satisfying the simulation criterion that involves multiple exit nodes n_i^x can be transformed back into a program which any reasonable nontermination insensitive criterion must accept, simply by replacing n_i^x with a loop, exiting into m_i (see, e.g., Figure 6.4b).

6.5 Soundness of Nontermination Insensitive Slices

So how can I characterize what nodes – in my opinion – are missing in $\rightarrow_{\text{wod}}$ slices? To do this, let me first recall (from [Amt08]) the notion and property of *next observable nodes* crucial for establishing the simulation property of $\rightarrow_{\text{wod}}, \rightarrow_{\text{data}}$ slices. That notion captures the following intuition: once an "unobservable region" is reached during execution, any decision made in this region (e.g., at nodes n within that region) shall have no influence on which other node is observed next when leaving this region — since otherwise, observing some node m instead of another (a priori possible) possible node m' will tell an observer which decision was made at n.

Definition 6.5.1 ([Amt08], Definition 3)**.** Given a CFG G, a node n, and a set S of nodes, the set $\text{obs}_S^G(n)$ of next-observable nodes is the set of nodes $m \in S$ with the property that in G there exists a non-empty path $\pi = n_1 \ldots, n_k$ with $n_1 = n$ and $n_k = m$ such that $n_i \notin S$ for $1 \leq i < k$.

Note that for $n \in S$, $\text{obs}_S^G(n) = \{n\}$.

Lemma 6.5.1 ([Amt08], Lemma 5)**.** Let G be any CFG, and assume that S is closed under $\rightarrow_{\text{wod}}$. Then for all n in G, $\text{obs}_S^G(n)$ is at most singleton.

This property is so crucial to the correctness of slicing that other authors, in effect[7], take it to be the *definition* of correctness.

Definition 6.5.2 (adapting [Dan+11], Definition 34 and Definition 9)**.** A node $n \notin S$ is S-weakly committing in G if $\text{obs}_S^G(n)$ is at most a singleton.

Definition 6.5.3 (adapting [Dan+11], Definition 35)**.** A set $S \subseteq N$ is weakly control closed in $G = (N, E)$ iff all vertices $n \notin S$ reachable from S are S-weakly committing in G.

[7] see: [Dan+11], Theorem 45

The additional new requirement I want to pose — which , in general, does *not* hold for sets S closed under $\rightarrow$wod (or, for that matter, about weakly control closed sets S) — is the following: Once execution reaches an S-unobservable region, no decision made in this region can influence whether the region can eventually be left, or not.

Formally — since obviously $\operatorname{obs}_S^G(n') \subseteq \operatorname{obs}_S^G(n)$ whenever $n \rightarrow_G n'$ for $n \notin S$ — I propose:

Definition 6.5.4. Given a CFG $G = (N, E)$, and a set of ("observable") nodes S, I say that a node $n \notin S$ *retains all possible next observations* iff

$$\operatorname{obs}_S^G(n) = \operatorname{obs}_S^G(n')$$

for all nodes n' such that $n \rightarrow_G n'$.

I say that *all possible next observations are retained outside* S iff this is the case for all $n \notin S$.

Recalling Figure 6.4a for the set $S = M = \{m_1, \ldots, m_k\}$ closed under $\rightarrow$wod, you will find that S is weakly control closed, but that *not* all possible next observations are retained outside S. For example, $\operatorname{obs}_S^G(n_1) = \{m_1\}$, but after the step $n_1 \rightarrow_G n_1^x$, I have $\operatorname{obs}_S^G(n_1) = \emptyset$.

On the other hand in Figure 6.4a for the same set $S = M = \{m_1, \ldots, m_k\}$, not only is S is weakly control closed, but also all possible next observations are retained outside S, since $\operatorname{obs}_S(n_i^\epsilon) = \operatorname{obs}_S(l) = \operatorname{obs}_S(m_i) = \{m_i\}$.

In general, the following holds:

Observation 6.5.1. If all possible next observations are retained outside S, then S is weakly control closed.

I propose that a slice S is to be deemed nontermination insensitively sound if all possible next observations are retained outside S. This criterion has a somewhat *syntactic* flavor: it does not explicitly reference some notion of execution of the graph G. Instead, it is directly stated

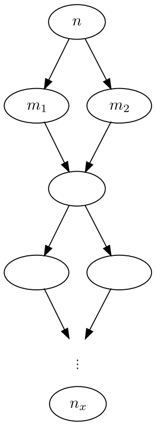


Figure 6.6: A CFG with infeasibly large $\rightarrow$wod.

in terms of reachability in the graph. Contrast this with Section 6.3. There, in the nontermination sensitive case, I gave a notion of input and executions (“traces”), and used a notion of trace-equivalence to give a soundness (and minimality) criterion for control slices. In other words, I gave a *semantic* criterion.

In an attempt to obtain a *semantic* criterion for control slices also in the nontermination *insensitive* case, in Section 6.6 I propose a similar, trace-based notion of execution in arbitrary graphs.

Then in Section 6.7, I will propose the new dependency notion $\rightarrow$ntiod. Slices S w.r.t. $\rightarrow$nticd, $\rightarrow$ntiod will not only be weakly control closed, but all possible next observations outside S will be retained, as well. They will also be minimal along all such slices. More importantly, though, $\rightarrow$ntiod will address the following criticism of $\rightarrow$wod as a foundation for practical slicing: Since it “covers” the whole CFG, and since it is a ternary relation, explicit representations of $\rightarrow$wod become infeasibly large even for graphs with unique exit node n_x. To get an Idea, consider Figure 6.6. Not only do I have $n \rightarrow_{\text{wod}} (m_1, m_2)$, but in fact $n \rightarrow_{\text{wod}} (m_1, m_2')$ for *all* $m_2' \notin \{n, m_1\}$.

6.6 A Trace-Based Notion of Infinite Delay

What is an appropriate "semantic" notion of nontermination insensitive slicing correctness? In this chapter, I will give a notion which I find satisfying for the "stateless" notion[8] of input and execution for CFG from Section 6.3. I consider this result preliminary, if only for the fact that it is not obvious to me how this notion can be generalized to a more traditional stateful notion of input and execution.

Reconsider Figure 6.4b. In the stateless notion of execution, consider inputs $i = (n_1, \epsilon)$ with entry node n_1. Then, if $\epsilon\left(n_j\right) = m_j$ for all j, the execution trace t_i is

$$(n_1, m_1),\ (m_1, n_2)\ ,\ldots,\ (n_k, m_k),\ (m_k, n_{k+1}),\ (n_{k+1}, \bot)$$

If, on the other hand, $\epsilon\left(n_j\right) = n^{\epsilon}_j$ for some j, the choice $\epsilon\left(n^{\epsilon}_j\right)$ made at n^{ϵ}_j determines whether an execution reaching n_j can continue towards m_j, or if it *infinitely delays* this continuation by choosing $\epsilon\left(n^{\epsilon}_j\right) = l_j$. Suppose, again, that $M = \{m_1, \ldots, m_k\}$ is the set of observable nodes. Then — as was the case for Figure 6.4a — any observable trace is of the form

$$t_i\big|_M = (m_1, n_2),\ \ldots,\ (m_j, n_{j+1})$$

with

$$\begin{array}{ll} j = k \text{ and} & t_i = \ldots,\ (m_j, n_{j+1}),\ \left(n_{k+1}, n^{\epsilon}_{k+1}\right) \\ \text{or} & t_i = \ldots,\ (m_j, n_{j+1}),\ \left(n_{j+1}, n^{\epsilon}_{j+1}\right),\ t,\ t,\ t,\ \ldots \end{array}$$

and

$$t = \left(n^{\epsilon}_{j+1}, l_{j+1}\right),\ \left(l_{j+1}, n^{\epsilon}_{j+1}\right)$$

[8] i.e.: with execution steps between configurations consisting only of nodes n instead of pairs (n, σ) with variable state σ.

Also: the only ("up to infinite delay") possible observable continuations of $t_i|_M$ are

$$\begin{aligned}
t'|_M &= t_i|_M\,,\ (m_{j+1}, n_{j+2}) \\
t''|_M &= t_i|_M\,,\ (m_{j+1}, n_{j+2})\,,\ (m_{j+2}, n_{j+3}) \\
&\ \vdots \\
&= t_i|_M\,,\ (m_{j+1}, n_{j+2})\,,\ (m_{j+2}, n_{j+3})\,,\ \ldots,\ (m_k, n_{k+1})
\end{aligned}$$

and form a prefix-chain

$$t_i|_M \sqsubseteq_{\mathcal{T}} t'|_M \sqsubseteq_{\mathcal{T}} t''|_M \sqsubseteq_{\mathcal{T}} \ldots$$

Let me — by example — explain how these infinitely delayed observations can be explained. Consider the observation $t'|_M$. It can be obtained as the concatenation of the finite observation $t_i|_M$ with an observation $t_{i'}|_M$ determined by an input $i' = (n'_e, \epsilon')$ which "breaks out of the loop": i' starts the execution at some node n'_e in the loop L which infinitely delays the observation (m_{j+1}, n_{j+2}), i.e.:

$$n'_e \in L = \left\{ n^{\epsilon}_{j+1},\ l_{j+1} \right\}$$

and then makes choices ϵ' *observably-consistent* with those made by i, i.e.

$$\epsilon'|_M = \epsilon|_M$$

For example:

$$\epsilon'(n) = \begin{cases} m_{j+1} & \text{for } n = n^{\epsilon}_{j+1} \\ n^{\epsilon}_{j+2} & \text{for } n = n_{j+2} \\ l_{j+2} & \text{for } n = n^{\epsilon}_{j+2} \\ \epsilon(n) & \text{otherwise} \end{cases}$$

where the first choice breaks out of the loop L, while the next two choices force the execution to remain in the *following* loop $L' = \{n^{\epsilon}_{j+2},\ l_{j+2}\}$.

Definition 6.6.1. Let G be any CFG, M be any set (of observable nodes), and $i = (n_e, \epsilon)$ any input to G. Then the set $\mathcal{T}^{\omega,G}_{i}\big|_M$ (or just $\mathcal{T}^{\omega}_{i}\big|_M$) of observable behavior up to infinite delay is defined to be

$$
\begin{cases}
\left\{ t_i\big|_M \right\} & \text{if } t_i \text{ is finite} \\[2ex]
\left\{ t_i\big|_M\, t_{i'}\big|_M \;\middle|\; i' = (n'_e, \epsilon')\,,\ \ \epsilon'\big|_M = \epsilon\big|_M,\ \ n'_e \in t \right\} & \text{if } t_i \text{ is infinite}
\end{cases}
$$

where for infinite t_i:

$$t_i = \left(n_e, n'\right),\ \ldots,\ t,\ t,\ t, \ldots$$

for some finite prefix followed by the infinitely repeating cycle t.

Obviously, $t_i\big|_M \in \mathcal{T}^{\omega}_{i}\big|_M$ and $t_i\big|_M \sqsubseteq_{\mathcal{T}} t$ for all $t \in \mathcal{T}^{\omega}_{i}\big|_M$.

Definition 6.6.2. Let G be any CFG, M be any set (of observable nodes), and i, i' two inputs G. Then input i, i are said to have M-equivalent observable behavior *up to infinite delay* iff the observable behavior up to infinite delay of i and i' form non-disjoint $\sqsubseteq_{\mathcal{T}}$-ascending chains.

Formally: $i \sim^{\omega}_{\mathcal{T}_M} i'$ iff

(a) $\mathcal{T}^{\omega}_{i}\big|_M = \{t_1, t_2, t_3, \ldots\}$ with $t_1 \sqsubseteq_{\mathcal{T}} t_2 \sqsubseteq_{\mathcal{T}} t_3 \sqsubseteq_{\mathcal{T}} \ldots$

(b) $\mathcal{T}^{\omega}_{i'}\big|_M = \{t'_1, t'_2, t'_3, \ldots\}$ with $t'_1 \sqsubseteq_{\mathcal{T}} t'_2 \sqsubseteq_{\mathcal{T}} t'_3 \sqsubseteq_{\mathcal{T}} \ldots$ and

(c) $\mathcal{T}^{\omega}_{i}\big|_M \cap \mathcal{T}^{\omega}_{i'}\big|_M \neq \emptyset$

The requirement item (c) appears to be remarkably permissive. Why can I not demand $\mathcal{T}^{\omega}_{i}\big|_M = \mathcal{T}^{\omega}_{i'}\big|_M$? Or, for example,

$$\mathcal{T}^{\omega}_{i}\big|_M \subseteq \mathcal{T}^{\omega}_{i'}\big|_M \quad \vee \quad \mathcal{T}^{\omega}_{i'}\big|_M \subseteq \mathcal{T}^{\omega}_{i}\big|_M$$

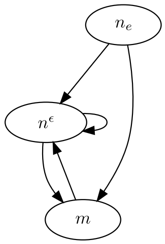


Figure 6.7: Infinite observable behavior up to infinite delay

To see why, consider Figure 6.7, for $M = \{m\}$, $i = (n_e, \epsilon)$, $i' = (n_e, \epsilon')$ and

$$\begin{aligned} \epsilon\,(n_e) &= n^\epsilon & \epsilon\,(n^\epsilon) &= n^\epsilon \\ \epsilon'\,(n_e) &= m & \epsilon'\,(n^\epsilon) &= n^\epsilon \end{aligned}$$

and thus

$$t_i\big|_M = \text{␣} \quad \text{and} \quad t_{i'}\big|_M = (m, n^\epsilon)$$

Intuitively, both inputs have just the same observable behavior up to infinite delay: an infinite sequence $t = (m, n^\epsilon),\ (m, n^\epsilon),\ \ldots$. But formally

$$\mathcal{T}_i^\omega\big|_M = \{t,\ \text{␣}\} \quad \text{and} \quad \mathcal{T}_{i'}^\omega\big|_M = \{t,\ (m, n^\epsilon)\}$$

Neither are these sets of observable traces the same, nor is one a subset of the other.

6.6.1 Other Criteria for Nontermination Insensitive Slicing

Aside from my two new notions (Definition 6.5.4 in Section 6.4 and Definition 6.6.2 in this section) and the simulation criterion (Section 6.4), several other approached defining correctness for nontermination insensitive slicing are possible.

In his thesis [Gif12], Dennis Giffhorn, too, proposes a notion of infinite delay in order to define low-equivalence between traces. Unlike the notions I developed in this section, Giffhorns definitions resort to the notion of dynamic control dependence, which, in turn resorts to (standard) control-dependence for CFG with unique end node. Hence, his notions *presuppose* that (dynamic) control dependence are appropriate to specify the meaning of *infinite delay*. I, on the other hand, developed the notions in this chapter in order to establish the adequacy of the notion of $\rightarrow_{\text{ntiod}}$ (to be defined in the following section) for the application of slicing of arbitrary CFG, and hence needed to give a criterion only based on observable traces $t_i|_M$.

Other notions are based on *transfinite* semantics of programs. In such semantics, for example, the program

```
x = 0;
while(true) {
 x = 1;
}
x = 2;
```

might be assigned the transfinite state-sequence

$$\underbrace{[x \mapsto 0]\,,\ [x \mapsto 1]\,,\ [x \mapsto 1]\,, \ldots,}_{\omega} [x \mapsto 2]$$

corresponding to the ordinal $\omega + 1$. Correctness of nontermination insensitive slicing then demands some correspondence between the transfinite semantics of the original and the transfinite semantics of the sliced program. In the example, slicing w.r.t `x` at the programs end and obtaining the program `x = 2` with the semantics $[x \mapsto 2]$ is considered correct. For details, see, e.g., [GM03], and [Bar+10] for a critique.
In fact, my notion $\mathcal{T}_i^{\omega}|_M$ of infinite delay can be understood as an ad-hoc “finitization” of a poor-mans transfinite semantic: instead of considering actual transfinite sequences, I consider all possibilities of short-cutting loops in and behind an infinite trace t_i with finite observation $t_i|_M$.

6.7 Nontermination Insensitive Order Dependence

In this section, I develop a nontermination *in*sensitive notion of order dependence. At first glance, this section is just a variation on Section 6.2, in which I developed a new notion of nontermination sensitive order dependence. Following the least-/greatest fixed point duality (Section B.2), I will mostly replace

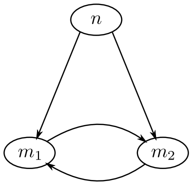
nontermination sensitivity	with	nontermination sensitivity
μ	with	ν
$\sqsupseteq_{\text{MAX}}$	with	$\sqsupseteq_{\text{SINK}}$
$>_{\text{MAX}}$	with	$>_{\text{SINK}}$
$\rightarrow_{\text{ntscd}}$	with	$\rightarrow_{\text{nticd}}$
$m_1 \sqsupseteq_{\text{MAX}[m_2]} n$	with	$m_1 \sqsupseteq_{\text{SINK}[m_2]} n$
$\rightarrow_{\text{ntsod}}$	with	$\rightarrow_{\text{ntiod}}$

However, there will also be the following differences w.r.t. Section 6.2:

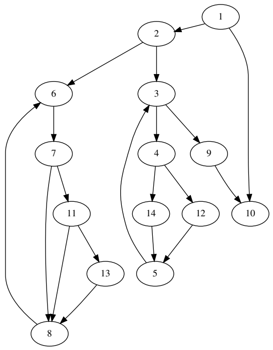
- The nodes n s.t. $n \rightarrow_{\text{ntiod}} (m_1, m_2)$ will *not* be restricted to nodes N_M "near the roots M of $<_{\text{SINK}}$", but also appear *within* those roots M s.t. $m_1, m_2 \in M$.
- It is *not* plausible to argue that such roots M (i.e.: such $<_{\text{SINK}}$ cycles) are, in practice, small. In fact, given any CFG G_0 with unique entry and exit nodes n_e, n_x, the graph G with edges $E = E_0 \cup (n_e, n_x)$ will yield a $>_{\text{SINK}}$-cycle $M = N$.
- Since with regards to the computation of $\rightarrow_{\text{ntiod}}$, I will ("again") have

$$\rightarrow_{\text{ntiod}} \quad \approx \quad \left(\sum_{M \in \mathbb{M}} |M| \right) \times \rightarrow_{\text{nticd}}$$

  I devise schemes that allow me — given $m_2 \in M$ and the postdominance-tree $>_{\text{SINK}}^{G_{m_2}}$ for one subgraph G_{m_2} — to compute



(a) Neither $n \rightarrow_{\text{nticd}} m_1$ nor $n \rightarrow_{\text{nticd}} m_2$.



(b) *Not:* $7 \rightarrow_{\text{nticd}} 11$ (nor $11 \rightarrow_{\text{nticd}} 13$, $7 \rightarrow_{\text{nticd}} 13$)

Figure 6.8: $\rightarrow_{\text{nticd}}$ is too small with respect to nontermination insensitive slicing

.

the postdominance-tree $>^{G_{m_2'}}_{\text{SINK}}$ for some other subgraph $G_{m_2'}$ *incrementally*, by (minimally) modifying $>^{G_{m_2}}_{\text{SINK}}$.

As argued in Section 6.4, I reject slicing w.r.t. $\rightarrow_{\text{wod}}$ on the grounds that such slices are *too small*, since outside such slices, in general, possible next observations are *not* retained. My goal can only be then to devise a new notion $\rightarrow_{\text{ntiod}}$ such that

$$(\rightarrow_{\text{nticd}} \cup \rightarrow_{\text{ntiod}})^* \quad \supseteq \quad (\rightarrow_{\text{wod}})^*$$

but not *much* larger.

The fact that *some* ternary relation is required can (as earlier for nontermination sensitive slicing) be demonstrated via the canonical irreducible CFG (Figure 6.8a), but also in the reducible CFG from Figure 6.8b. Specifically, $\rightarrow_{\text{nticd}}$ is too small, but even more: *no* binary (dependence) relation $\rightarrow_{\text{xcd}}$ precisely characterizes nontermination sensitive slicing in those CFG. Consider Figure 6.8b. Intuitively, I would want $7 \rightarrow_{\text{xcd}} 11$. But if 11 is the *only* observable node, then *all* observable traces reaching 6 are of the form $11, 11, \ldots$, and hence a slice including 7 would be imprecise. On the other hand I would *need* something like $7 \rightarrow_{\text{xcd}} 11$ as soon as, for example, node 8 is also observable.

Observation 6.7.1. Let $G = (N, E)$ be either of the CFG from Figure 6.8 (or: the subgraph $G|_{\{6,7,8,11,13\}}$ of Figure 6.8b). Then there exists no binary relation $\rightarrow_{\text{xcd}}$ such that for all $M \subseteq N$

$$(\rightarrow_{\text{xcd}})^* (M) = (\rightarrow_{\text{wod}})^* (M)$$

The same observation will hold for $\rightarrow_{\text{wod}}$ replaced with $\rightarrow_{\text{nticd}} \uplus \rightarrow_{\text{ntiod}}$.

Definition 6.7.1 ($\rightarrow_{\text{ntiod}}$, corresponding to Definition 6.2.1)**.** Let G be any CFG, and n, m_1, m_2 be distinct nodes. Then m_1 is **n**on**t**ermination **i**nsensitively **o**rder **d**ependent on n with respect to m_2, written: $n \rightarrow_{\text{ntiod}} (m_1, m_2)$, iff

(a) All sink paths from n contain both m_1 and m_2, i.e.:

$$m_1 \sqsupseteq_{\text{SINK}} n \quad \text{and} \quad m_2 \sqsupseteq_{\text{SINK}} n$$

(b) There exists some successor n_l of n such that all sink paths π_l starting in n_l contain m_1 before any occurrence of m_2, i.e.:

$$\exists n \rightarrow_G n_l. \quad \forall \pi \in {}_{n_l}\Pi_{\text{SINK}}.\ m_1 \sqsupseteq_{m_2} \pi$$

(c) There exists some successor n_r of n such that *not* all sink paths π_r starting in n_r contain m_1 before any occurrence of m_2, i.e.:

$$\exists n \rightarrow_G n_r.\ \neg\ \forall \pi \in {}_{n_r}\Pi_{\text{SINK}}.\ m_1 \sqsupseteq_{m_2} \pi$$

Again (and as opposed to $\rightarrow_{\text{wod}}$), this definition is *not* symmetric in m_1, m_2. Also note again that after holding fast m_2, and subject to the constraint (a), the relation $\cdot \rightarrow_{\text{ntiod}} (\cdot, m_2)$ follows the scheme of $\sqsupseteq$ control dependence (see Definition 3.1.2).

Definition 6.7.2 ($m_1 \sqsupseteq_{\mathsf{SINK}[m_2]} n$, corresponding to Definition 6.2.2)**.** Let G be some CFG.

$$m_1 \sqsupseteq^G_{\mathsf{SINK}[m_2]} n \quad \Leftrightarrow \quad \forall \pi \in {}_n\Pi_{\mathsf{SINK}}.\ m_1 \sqsupseteq_{m_2} \pi$$

As expected, $\sqsupseteq_{\mathsf{MAX}[m_2]}$ is not the least but the greatest fixed point of a suitable rule system.

Proposition 6.7.1. Let G be a CFG, m_2 any of its nodes, and D_{m_2} be the rule-system from Proposition 6.2.1 on page 73. Then $\sqsupseteq_{\mathsf{SINK}[m_2]} = \nu \mathsf{D}_{m_2}$.

Comparison with Weak Order Dependence

The relation $\rightarrow_{\text{ntiod}}$ (together with $\rightarrow_{\text{nticd}}$) is meant to replace $\rightarrow_{\text{wod}}$. In fact, it is a subset:

Observation 6.7.2. Let G be any CFG. Then

$$n \rightarrow_{\text{ntiod}} (m_1, m_2) \quad \Rightarrow \quad n \rightarrow_{\text{wod}} (m_1, m_2)$$

Remember that $\rightarrow_{\text{wod}}$ "covers" the whole CFG, while $\rightarrow_{\text{ntiod}}$ is meant to "cover" only those regions not covered by $\rightarrow_{\text{nticd}}$, i.e.: the control sinks. Also remember that $\rightarrow_{\text{wod}}$ is symmetric, while $\rightarrow_{\text{ntiod}}$ is not.

In the other direction the situation is as follows:

Observation 6.7.3. Let G be any CFG, and assume

$$n \rightarrow_{\text{wod}} (m_1, m_2)$$

Then

$$\begin{aligned} & n \rightarrow_{\text{ntiod}} (m_1, m_2) \quad \vee \quad n \rightarrow^*_{\text{nticd}} m_1 \\ \vee \quad & n \rightarrow_{\text{ntiod}} (m_2, m_1) \quad \vee \quad n \rightarrow^*_{\text{nticd}} m_2 \end{aligned}$$

Corollary 6.7.1.

$$(\to_{\text{wod}})^* \subseteq (\to_{\text{nticd}} \mathbin{\dot\cup} \to_{\text{ntiod}})^*$$

For example in Figure 6.4a on page 87 and with $M = \{m_1, \ldots\}$, the slice $(\to_{\text{wod}})^* (M)$ contains only the nodes M, while the slice $(\to_{\text{nticd}} \mathbin{\dot\cup} \to_{\text{ntiod}})^* (M)$ additionally contains the nodes $\{n_1, \ldots\}$. This is required by Definition 6.5.4, since if these nodes were missing, the slice would not retain next-observable nodes. Similarly in Figure 6.8b and with $M = \{8, 11\}$, the slice $(\to_{\text{wod}})^* (M)$ contains M and node 7, while $(\to_{\text{nticd}} \mathbin{\dot\cup} \to_{\text{ntiod}})^* (M)$ additionally contains the nodes $\{1, 2\}$.

Structural Properties

In support of algorithms for nontermination insensitive order dependence $\to_{\text{ntiod}}$, I now characterize the regions in which $\to_{\text{ntiod}}$ is non-empty.

Observation 6.7.4 (corresponding to a weakening of Lemma 6.1.2)**.** Let G be any CFG.

(i) Whenever $n \to_{\text{ntiod}} (m_1, m_2)$, then

$$m_1 \sqsupseteq_{\text{SINK}} m_2 \quad \text{and} \quad m_2 \sqsupseteq_{\text{SINK}} m_1$$

(ii) Whenever $n \to_{\text{ntiod}} (m_1, m_2)$, and $m \sqsupseteq_{\text{SINK}} n$ for $m \neq n$. Then

$$m_1 \sqsupseteq_{\text{SINK}} m \quad \text{and} \quad m \sqsupseteq_{\text{SINK}} m_1$$

as well as

$$m_2 \sqsupseteq_{\text{SINK}} m \quad \text{and} \quad m \sqsupseteq_{\text{SINK}} m_2$$

As shown before, $\sqsupseteq_{\text{SINK}}$ can be efficiently computed and represented as transitive reduction $>_{\text{SINK}}$, which turned out to be a pseudo-forest.

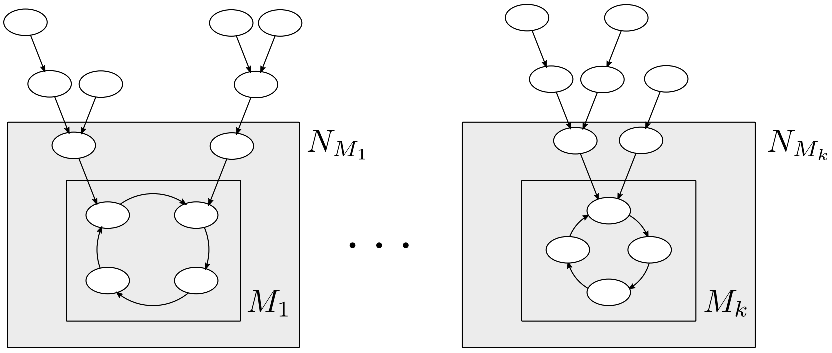


Figure 6.9: The situation in Corollary 6.7.2, exemplified. Given $m_1, m_2 \in M_i$, the set of n such that $n \rightarrow_{\text{ntiod}} (m_1, m_2)$ is contained in N_{M_i}.

The situation in Observation 6.7.4, rephrased in terms of $>_{\text{SINK}}$ — is depicted in Figure 6.9, and formally reads:

Corollary 6.7.2 (corresponding to a weakening of Corollary 6.1.1)**.** Let G be any CFG, and $>_{\text{SINK}}$ any transitive reduction of $\sqsupseteq_{\text{SINK}}$.

For $n \rightarrow_{\text{ntiod}} (m_1, m_2)$, let $M = \{\, m \mid m_1 <^*_{\text{SINK}} m \,\} = \{\, m \mid m_2 <^*_{\text{SINK}} m \,\}$ be the $<_{\text{SINK}}$-cycle both m_1, m_2 are part of. Then

- $n <_{\text{SINK}} m$ for some $m \in M$.

6.7.1 An Incremental Algorithm Based on Nontermination Insensitive Postdominance

Owing to Observation 6.7.4, I can limit the search for triples $n \rightarrow_{\text{ntiod}} (m_1, m_2)$ the same way I previously did for $\rightarrow$dod. But, just as was the case for $\rightarrow$ntsod, I can do more: for fixed m_2, the relation $\cdot \rightarrow_{\text{ntiod}} (\cdot, m_2)$ is essentially $\sqsupseteq_{\text{SINK}[m_2]}$-control dependence, with $\sqsupseteq_{\text{SINK}[m_2]}$ *almost* equal to $\sqsupseteq_{\text{SINK}}$: I have

$$\begin{array}{rlcl} & \forall \pi \in {}_{n_s}\Pi_{\text{SINK}}.\ m_1 \sqsupseteq_{m_2} \pi & \Leftrightarrow & m_1 \sqsupseteq_{\text{SINK}[m_2]} n_s \\ \text{instead of} & \forall \pi \in {}_{n_s}\Pi_{\text{SINK}}.\ m_1 \in \pi & \Leftrightarrow & m_1 \sqsupseteq_{\text{SINK}} \quad n_s \end{array}$$

Again, by choosing — given m_2 — a suitable subgraph G, it is not hard to reduce $\sqsupseteq_{\text{SINK}[m_2]}$ in G to $\sqsupseteq_{\text{SINK}}$ in the subgraph.

Observation 6.7.5 (corresponding to Observation 6.2.4)**.** Let $G = (N, E)$ be any CFG, and $n_s, m_1, m_2 \in N$, $m_1 \neq m_2$. Then any G-sink-path starting in n_s contains m_1 before any occurrence of m_2 iff *in the graph $G_{m_2 \not\rightarrow}$ obtained by removing all outgoing edges of m_2*, any sink-path starting in n_s contains m_1, i.e.:

$$m_1 \sqsupseteq^{G}_{\text{SINK}[m_2]} n_s \quad \Longleftrightarrow \quad m_1 \sqsupseteq^{G_{m_2 \not\rightarrow}}_{\text{SINK}} n_s$$

Again, combining Observation 6.7.4 and Observation 6.7.5 in a form suggesting an algorithm, I make the following observation:

Observation 6.7.6 (corresponding to Observation 6.2.5)**.** Let $G = (N, E)$ be any CFG, $> \; = \; >_{\text{SINK}}$ any transitive reduction of $\sqsupseteq^{G}_{\text{SINK}}$, $\mathbb{M}$ the set of $<$-cycles (i.e.: the set of control sinks), and let for $M \in \mathbb{M}$ the set of conditional nodes n s.t. $n < m$ for some $m \in M$ be denoted by N_M.

Then, $n \rightarrow_{\text{ntiod}} (m_1, m_2)$ if and only if in the CFG

$$G_{m_2} := G^{N_M \rightarrow^* M}_{m_2 \not\rightarrow}$$

$m_1 \neq n$ is $\rightarrow_{\text{nticd}}$-control dependent on n, where $M \in \mathbb{M}$ is the $<$-cycle containing both m_1 and m_2, i.e.:

$$\begin{aligned} n \rightarrow_{\text{ntiod}} (m_1, m_2) \Leftrightarrow \; & \exists M \in \mathbb{M}. m_1, m_2 \in M \;\wedge\; n \in N_M \;\wedge\; n \rightarrow^{G_{m_2}}_{\text{nticd}} m_1 \\ & \wedge\; m_1 \neq n \end{aligned}$$

Informally:

$$\to_{\text{ntiod}} \quad\approx\quad \left(\sum_{M\in\mathbb{M}} |M|\right) \times \to_{\text{nticd}}$$

Note that in Observation 6.2.5, $m_1 \neq n$ was implied by $m_1 \in M$, but this is not the case here, since here N_M, does not exclude M.

As argued in the opening words of this section, it is not unreasonable to expect large $M \in \mathbb{M}$, e.g.: $M = N$. A naive algorithm will simply compute $\to_{\text{nticd}}$ in G_m for each sink-node m, by computing $<_{\text{SINK}}$ and then $\to_{\text{nticd}}$ from scratch for each such graph. In order to achieve a reasonable performance for graphs with big $M \in \mathbb{M}$, I will replace the from-scratch $<_{\text{SINK}}$ computations by incremental computations.

Specifically, for each $M \in \mathbb{M}$, I will compute the

$$<_m \quad=\quad <^{G_m}_{\text{SINK}}$$

such that $>_m$ is a transitive reduction of $\sqsupseteq^{G_m}_{\text{SINK}}$, i.e.:

$$\sqsupseteq_m \quad=\quad \sqsupseteq^{G_m}_{\text{SINK}}$$

for

$$\begin{aligned} \sqsupseteq_m &= (>_m)^* \\ G_m &= (G_M)_{m\not\to} \quad \text{and} \\ G_M &= G^{N_M \to^* M} \end{aligned}$$

for each $m \in M$ incrementally.

By construction, the following holds:

Proposition 6.7.2.

1. G_m is a graph with unique exit node m.
2. $<_m$ is a tree with root m (i.e.: $\forall n \neq m.\ \exists! n'.\ n <_m n'$, and $\neg\, \exists n.\ m <_m n$).

Observation 6.7.7. Let M be any control-sink of a CFG G (and hence: a $>^{G}_{\text{SINK}}$-cycle), and use the above notation for any node $m \in M$. Let $m_2, m_2' \in M, m_2 \neq m_2'$ such that $m_2 \rightarrow_{G_M} m_2'$. Furthermore, let

$$\begin{array}{rcl} <'_{m_2} & = & \{\, (n, m\ ) \mid n <_{m_2} m,\ n \neq m_2',\ \neg n \rightarrow_{G_M} m_2' \,\} \\ & \cup & \{\, (n, m_2') \mid \qquad\qquad n \neq m_2', \quad n \rightarrow_{G_M} m_2' \,\} \end{array}$$

be the tree which is obtained from $<_{m_2}$ by making m_2' the new root, and letting all G_M-predecessors $n \neq m_2'$ of m_2' point to m_2', and write $\sqsupseteq'_{m_2}$ for $\left(>'_{m_2}\right)^*$. Then the following holds:

1. If m_2 is the only G_M-predecessor of m_2', then $>'_{m_2}$ is a transitive reduction of $\sqsupseteq^{G_{m_2'}}_{\text{SINK}}$, i.e.:

$$\sqsupseteq'_{m_2} \quad = \quad \sqsupseteq_{m_2'}$$

2. Otherwise, $\sqsupseteq'_{m_2}$ approximates $\sqsupseteq^{G_{m_2'}}_{\text{SINK}}$ from above, i.e.:

$$\sqsupseteq'_{m_2} \quad \supseteq \quad \sqsupseteq_{m_2'}$$

   Also, if I write $<_0$ for the two-tree forest $(<_{m_2})_{m_2' \not\rightarrow}$, then for any nodes n such that $n \rightarrow_{G_M} m_2'$ or $\neg\ n <_0^* m_2$, I have:

$$\{\, x \mid x \sqsupseteq'_{m_2} n \,\} \quad = \quad \{\, x \mid x \sqsupseteq_{m_2'} n \,\}$$

I attempt to visualize this process in figure Figure 6.10.

With regard to computation costs, I note the following:

- The computation of $<'_{m_2}$ from $<_{m_2}$ is trivially cheap for any but the most ridiculous graphs G.
- The fact that $\sqsupseteq'_{m_2}$ is a superset of $\sqsupseteq_{m_2'}$ allows me to compute $\sqsupseteq_{m_2'}$ by immediately starting a fixed-point computation from

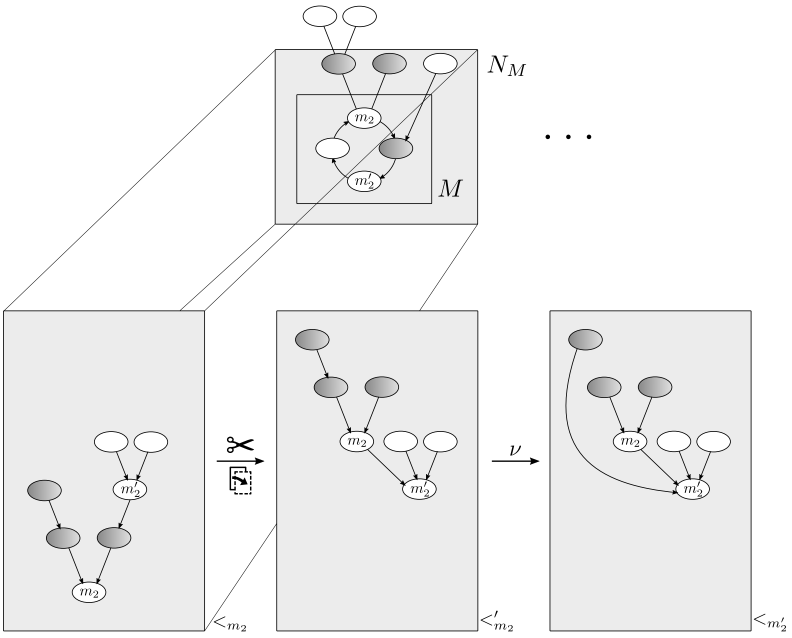


Figure 6.10: The Process in Observation 6.7.7. On top: $<^G_{\text{SINK}}$. The last step is only necessary if m_2 is not the only predecessor of m_2'.

$\sqsupseteq'_{m_2}$ (e.g., by using phase SINK$_{\text{down}}$ from Algorithm 6). I do not need to execute phase SINK$_{\text{up}}$.

- Even more, I am done for any node n such that $n \rightarrow_{G_M} m_2'$ or $\neg\ n <^*_0 m_2$, and hence can modify workset-based algorithms such as SINK$_{\text{down}}$ to never put such nodes in the workset!

Now, Observation 6.7.7 only applies whenever $m_2 \rightarrow_{G_M} m_2'$, but this already allows me to compute $\sqsupseteq_m$ for all $m \in M$ as follows:

1. Using any heuristic, enumerate a sequence $\pi_1, \ldots, \pi_k$ of (finite) G_M-paths in M such that

   $$M = \bigcup_i \pi_i$$

2. For all

   $$\pi_i = m, \ldots$$

   compute $\sqsupseteq_m$ using any postdominator-tree algorithm (remember that G_m is a CFG with unique exit node m). Then, for all other nodes $m_2' \in \pi_i$ with π_i-predecessor m_2, compute $\sqsupseteq_{m_2'}$ using $\sqsupseteq_{m_2}$ via Observation 6.7.7.

Any heuristic that chooses the π_i should then, presumably:

- Attempt to cover each node $m \in M$ only once.
- Try to minimize the number of π_i-neighbours m_2, m_2' such that m_2 is not the only G_M-predecessor of m_2'.
- Try to minimize the number k of π_i, since every $\pi_i = m, \ldots$ requires an initial full computation of $\sqsupseteq_m$.

My implementation uses a greedy heuristic that finds paths starting in join nodes, and covers each node exactly once.

A Variant for Arbitrary Enumeration of Sink Nodes

Owing to the following observation, it turns out that the full computations of $\sqsupseteq_m$ for $\pi_i = m, \ldots$ is not even necessary. I attempt to visualize the corresponding process in figure Figure 6.11 on page 112.

Observation 6.7.8. Let M be any control-sink of a CFG G (and hence: a $>^{G}_{\text{SINK}}$-cycle), and use the above notation for any node $m \in M$. Let $m_2 \neq m_2'$ be *any* nodes in M. Furthermore, let

$$<_0 \quad = \quad (<_{m_2})_{m_2' \not\to}$$

be the two-tree forest obtained by cutting the subtree with root m_2' from the tree $<_{m_2}$ rooted in m_2, and write $\sqsupseteq_0$ for $(>_0)^*$.

Also, let

$$P_{m_2'} = \{\, n \mid \qquad m_2' \sqsupseteq_0 n \,\}$$

be the set of nodes n that are in the tree rooted in m_2',

$$S_{m_2'} = \{\, n \mid m_2 \to_G n,\ m_2' \sqsupseteq_0 n \,\}$$

be the set of G-successors n of m_2 that are in the tree rooted in m_2',

$$C_{m_2} = \{\, n \mid n \in \text{COND}_{G_{m_2'}},\ m_2 \sqsupseteq_0 n \,\}$$

be the set of conditional (in $G_{m_2'}$) nodes n that are in the tree rooted in m_2.

Then

1. If $S_{m_2'}$ is non-empty, there is a unique least common ancestor $z \in \text{lca}_{<_0}\left(S_{m_2'}\right)$ of all such $S_{m_2'}$, and the graph

$$<'_{m_2} \quad = \quad <_0 \ \cup\ \{(m_2, z)\}$$

   is a tree (rooted in m_2'); specifically — in the words of *Algorithm* 6 — every node is *processed*.

2. If $S_{m_2'}$ is empty, then for all $n \in P_{m_2'}$:

$$\{\, x \mid x \sqsupseteq_0 n \,\} \quad \supseteq \quad \{\, x \mid x \sqsupseteq_{m_2'} n \,\}$$

Also, let

$$<'_0 \;=\; \{\, (n, m\,) \mid n <_0 m,\ n \notin C_{m_2} \,\} \cup \begin{cases} \emptyset & \text{if } m_2 \in \text{COND}_{G_{m'_2}} \\ \{(m_2, m)\} & \text{if } m \text{ is the unique } G_{m'_2} - \text{successor of } m_2 \end{cases}$$

and let $<'_{m_2}$ be the tree obtained by "processing" the tree $<'_0$, i.e.: let $<'_{m_2}$ be the tree obtained by running phase SINK$_{\text{up}}$[9] from *Algorithm* 6 with initial set $\mathsf{PROCD} = P_{m'_2}$ of processed nodes, and initial $\mathsf{workqueue} = C_{m_2}$.

In both cases, write $\sqsupseteq'_{m_2}$ for $\left(>'_{m_2}\right)^*$. Then:

$$\sqsupseteq'_{m_2} \quad \supseteq \quad \sqsupseteq_{m'_2}$$

and also for $n \notin \{\, n \mid \ m_2 \sqsupseteq_0 n \,\} \supseteq C_{m_2}$:

$$\{\, x \mid x \sqsupseteq'_{m_2} n \,\} \quad = \quad \{\, x \mid x \sqsupseteq_{m'_2} n \,\}$$

With regard to computation costs, I note the following:

- The computation of $<'_{m_2}$ from $<_{m_2}$ is trivially cheap if $S_{m'_2} \neq \emptyset$, but requires a partial execution of phase SINK$_{\text{up}}$ if it is empty.
- Again, the fact that $\sqsupseteq'_{m_2}$ is a superset of $\sqsupseteq_{m'_2}$ allows me to compute $\sqsupseteq_{m'_2}$ by immediately starting a fixed-point computation from $\sqsupseteq'_{m_2}$ (e.g., by using phase SINK$_{\text{down}}$ from Algorithm 6).
- Even more, I am done for any node $n \notin C_{m'_2}$, and hence can modify workset-based algorithms such as SINK$_{\text{down}}$ to never put such nodes in the workset!

Thanks to Observation 6.7.8 — together with Observation 6.7.7 — I can compute $\sqsupseteq_m$ for all $m \in M$ as follows:

[9] for $G_{m'_2}$

1. Using any heuristic, enumerate any finite sequence[10] π of nodes such that $M = \{ m \mid m \in \pi \}$
2. For
$$\pi = m, \ldots$$
compute $\sqsupseteq_m$ using any postdominator-tree algorithm. Then, for all other nodes $m_2' \in \pi$ with π-predecessor m_2, compute $\sqsupseteq_{m_2'}$ using $\sqsupseteq_{m_2}$ via Observation 6.7.7 if $m_2 \rightarrow_{G_M} m_2'$, or via Observation 6.7.8 if not.

Any heuristic that chooses π should then, presumably:

- Attempt to cover each node $m \in M$ only once.
- Try to minimize the number of π-neighbours m_2, m_2' such that m_2 is not a G_M-predecessor of m_2', and then
- try to minimize the number of π-neighbours m_2, m_2' such that m_2 is not the only G_M-predecessor of m_2'.

Using this scheme, and taking into account Observation 6.7.6, the computation of $\rightarrow_{\text{ntiod}}$ for a CFG G reduces to the computation of

1. $<^G_{\text{SINK}}$
2. the set $\mathbb{M}$ of control-sinks M of G[11]
3. for each $M \in \mathbb{M}$ the subgraph G_M
4. for each $m_2 \in M$:
    a) $<_m$ as described
    b) the set of $n \in N_M$, $m_1 \in M$, $m_1 \neq n$ such that $n \rightarrow^{G_{m_2}}_{\text{nticd}} m_1$, obtaining $n \rightarrow_{\text{ntiod}} (m_1, m_2)$.

[10] not necessarily a G_M-path!
[11] which are exactly the $<^G_{\text{SINK}}$-cycles, and also: exactly the strongly connected components of G that do not have edges leaving the component.

The computation item 4b can be done using any postdominance frontier algorithm.

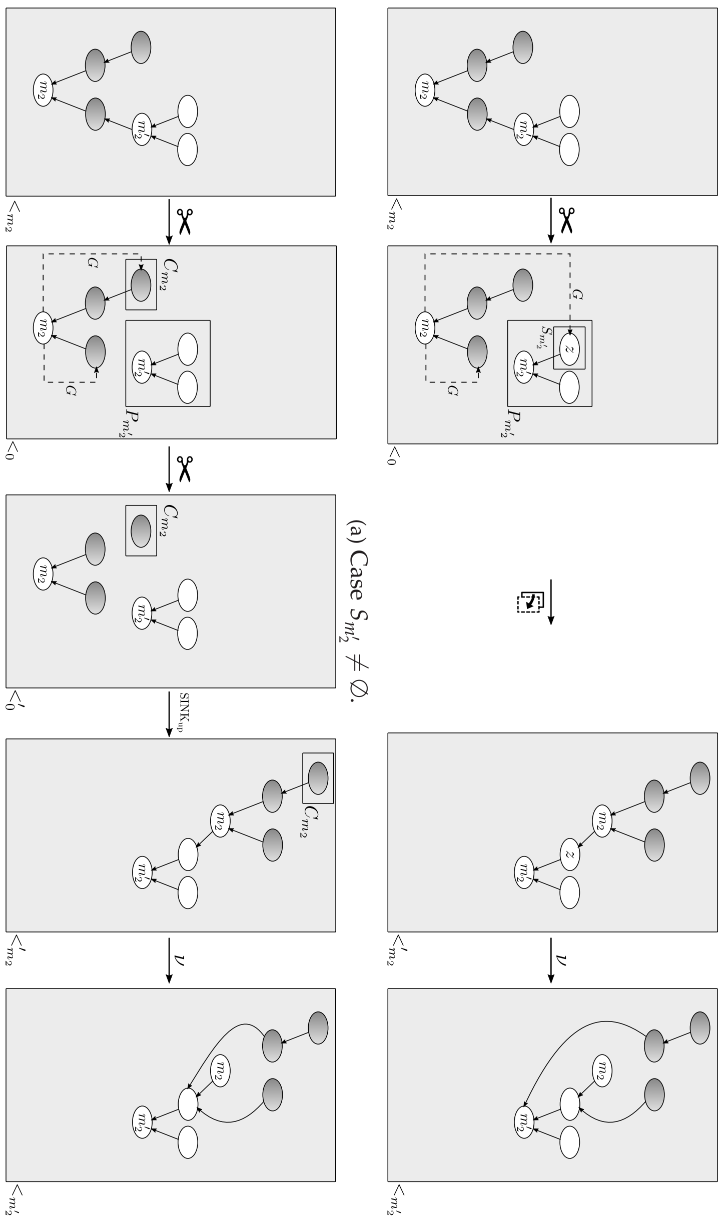
Remark 6.7.1. I described a scheme to obtain

$$<_{m_2'} \text{ from } <_{m_2}$$

I did *not* investigate whether it is possible to directly obtain

$$\rightarrow_{\text{nticd}}^{G_{m_2'}} \text{ from } \rightarrow_{\text{nticd}}^{G_{m_2}}$$

Although this might very well be possible, it appears to be of limited practical value, since experiments suggest that in practice, the scheme I presented already yields an algorithm with execution time roughly linear in the size of the relation $\rightarrow_{\text{ntiod}}$.



(a) Case $S_{m_2'} \neq \emptyset$.

(b) Case $S_{m_2'} = \emptyset$.

Figure 6.11: The Process in Observation 6.7.7. Dashed arrows are edges in G.

6.8 Soundness of Nontermination Insensitive Order Dependence

In Section 6.4, I recalled the simulation based correctness criterion for nontermination insensitive slicing which is satisfied by $\rightarrow_{\text{wod}}$ ([Amt08]). I offered a critique of this criterion by example of an "always terminating" CFG that satisfies the simulation-criterion for some set $S = M$ of observable nodes, but intuitively leaks information.

The simulation criterion is based on "stateful" executions of CFG with configurations as pairs (n, σ) of nodes and variable assignments. It requires closure under $\rightarrow_{\text{data}}$. In [Amt08], it was shown that S being weakly control-closed is useful in proving the simulation property. In [Was10], for graphs with unique end nodes n_x, and assuming $n_x \in S$, it was shown that being weakly control-closed is outright the only property of control-dependency necessary to establish the simulation property, and in [Dan+11] this property is straight out taken to *define* correctness of CFG slicing.

While I did not propose a strengthened version of the simulation criterion, I *did* (in Section 6.5) propose a new requirement for (unlabeled) CFG (in addition to being weakly control-closed) that prevents information leak for always-terminating programs. I also proposed — in Section 6.6 — a new trace-based criterion with the same purpose.

In this section, I confirm that $\rightarrow_{\text{ntiod}}$[12] is, indeed, both sound and minimal with respect to these two new criteria. I also confirm that, for always-terminating CFG, slicing w.r.t $\rightarrow_{\text{ntiod}}$ is nontermination *sensitively* sound, i.e.: observable traces are uniquely defined by the observable input, which is *not* the case for slicing w.r.t $\rightarrow_{\text{wod}}$.

Observation 6.8.1 (Soundness of $\rightarrow_{\text{ntiod}}, \rightarrow_{\text{nticd}}$ w.r.t obs_S)**.** Let G be any CFG, and $M \subseteq N$ a set of nodes (the slicing criterion). Let $S = (\rightarrow_{\text{nticd}} \cup\!\!\!\cdot\; \rightarrow_{\text{ntiod}})^* \,(M)$ be the backward slice w.r.t. M. Then all

[12] together with $\rightarrow_{\text{nticd}}$

possible next observations are retained outside S (see Definition 6.5.4 on page 90).

Observation 6.8.2 (Minimality of $\rightarrow_{\text{ntiod}}, \rightarrow_{\text{nticd}}$ w.r.t obs_S)**.** Let G be any CFG, and $M \subseteq N$ a set of nodes (the slicing criterion). Let $S = (\rightarrow_{\text{nticd}} \cup \rightarrow_{\text{ntiod}})^* (M)$ be the backward slice w.r.t. M. Then, for any $n \in S, n \notin M$ and $S' = S \setminus \{n\}$, *not* all possible next observations are retained outside S'

Minimality of $\rightarrow_{\text{ntiod}}$ as stated in Observation 6.8.2 is directly amenable to empirical validation. For the following alternative notion of minimality, Simon Bischof prepared a machine checked proof in the Isabelle/HOL proof assistant:

Theorem 6.8.1 ([Bis19], Minimality of $\rightarrow_{\text{ntiod}}, \rightarrow_{\text{nticd}}$ w.r.t obs_S)**.** Let G be any CFG, and $M \subseteq N$ a set of nodes (the slicing criterion). Let $S = (\rightarrow_{\text{nticd}} \cup \rightarrow_{\text{ntiod}})^* (M)$ be the backward slice w.r.t. M. Let $S' \supseteq M$ be any set of nodes that retains all possible next observations. Then $S' \supseteq S$.

Slicing via $\rightarrow_{\text{ntiod}}$ and $\rightarrow_{\text{nticd}}$ is also sound and minimal with regard to the nontermination insensitive trace based criterion of *infinite delay*.

Observation 6.8.3 (Soundness of $\rightarrow_{\text{ntiod}}, \rightarrow_{\text{nticd}}$ w.r.t $\mathcal{T}^{\omega}\big|_S$)**.** Let G be any CFG, and $M \subseteq N$ a set of nodes (the slicing criterion). Let $S = (\rightarrow_{\text{nticd}} \cup \rightarrow_{\text{ntiod}})^* (M)$ be the backward slice w.r.t. M. Then, any S-equivalent inputs i, i' — i.e.: inputs such that

$$i \sim_S i'$$

— have S-equivalent observable behavior *up to infinite delay*[13], i.e.:

$$i \sim^{\omega}_{\mathcal{T}_S} i'$$

[13] see Definition 6.6.2 on page 94

Observation 6.8.4 (Minimality of $\rightarrow_{\text{ntiod}}, \rightarrow_{\text{nticd}}$ w.r.t $\mathcal{T}^{\omega}\big|_S$)**.** Let G be any CFG, and $M \subseteq N$ a set of nodes (the slicing criterion). Let $S = (\rightarrow_{\text{nticd}} \,\uplus\, \rightarrow_{\text{ntiod}})^* (M)$ be the backward slice w.r.t. M. Then, for any $n \in S, n \notin M$ and $S' = S \setminus \{n\}$, there exists S'-equivalent inputs i, i' — i.e.: inputs such that

$$i \sim_S i'$$

— that do *not* have S'-equivalent observable behavior *up to infinite delay*, i.e.:

$$\neg\, i \sim^{\omega}_{\mathcal{T}_S} i'$$

The two preceding observation are the nontermination *in*sensitive analogues of observations 6.3.1 and 6.3.2 that concerned nontermination *sensitive* slicing. The following observation confirms that, for always-terminating CFG, nontermination insensitive slicing is as strict as nontermination sensitive slicing.

Observation 6.8.5 (Trace Equivalence for $\rightarrow_{\text{ntiod}}, \rightarrow_{\text{nticd}}$ in acyclic CFG)**.** Let G be a CFG in which *all executions terminate*, i.e.: an acyclic graph. Let $M \subseteq N$ be a set of nodes (the slicing criterion), and let $S = (\rightarrow_{\text{nticd}} \,\uplus\, \rightarrow_{\text{ntiod}})^* (M)$ be the backward slice w.r.t. M. Then, any S-equivalent inputs i, i' — i.e.: inputs such that

$$i \sim_S i'$$

— have nontermination sensitively S-equivalent traces[14], i.e.:

$$t_i \sim_S t_{i'}$$

Remember that this last observation, although relatively weak, does *not* hold for slicing under $\rightarrow_{\text{wod}}$, as was exemplified in Figure 6.4a on page 87.

[14] see Definition 6.3.4 on page 80

Summary

- Together, decisive order dependence $\rightarrow_{\text{dod}}$ and nontermination sensitive control dependence $\rightarrow_{\text{ntscd}}$ produce sound and minimal slices.
- The same holds for nontermination insensitive order dependence $\rightarrow_{\text{ntsod}}$.
- $\rightarrow_{\text{ntsod}}$ can naturally be reduced to $\rightarrow_{\text{ntscd}}$, which leads to an algorithm for $\rightarrow_{\text{ntsod}}$.
- Together, nontermination sensitive order dependence $\rightarrow_{\text{ntiod}}$ and nontermination insensitive control dependence $\rightarrow_{\text{nticd}}$ produce sound and minimal slices.
- Soundness and minimality of nontermination insensitive slicing can either be defined by a notion of *infinite delay*, or by a condition of *next observables*.
- $\rightarrow_{\text{ntiod}}$ can naturally be reduced to $\rightarrow_{\text{nticd}}$, which leads to an algorithm for $\rightarrow_{\text{ntiod}}$.

7 Slicing

If you do something too good, then, after a while, if you don't watch it, you start showing off.

(J.D. Salinger — The Catcher in the Rye)

In the preceding chapters, I devised new algorithms for the computation of (explicit representations of) the dependency relations

1. **N**on-**T**ermination **S**ensitive **C**ontrol **D**ependence $\rightarrow_{\text{ntscd}}$
   a) indirectly via $<_{\text{MAX}}$ and Algorithm 1
2. **N**on-**T**ermination **I**nsensitive **C**ontrol **D**ependence $\rightarrow_{\text{nticd}}$
   a) directly via Algorithm 16
   b) indirectly via $<_{\text{SINK}}$ and Algorithm 1
3. **D**ecisive[1] **O**rder **D**ependence $\rightarrow_{\text{dod}}$
   a) via the generate-and-test scheme 6.3 on page 70
4. **N**on-**T**ermination **S**ensitive **O**rder **D**ependence $\rightarrow_{\text{ntsod}}$
   a) via the scheme implied by Observation 6.2.3 and Observation 6.2.5
5. **N**on-**T**ermination **I**nsensitive **O**rder **D**ependence $\rightarrow_{\text{ntiod}}$
   a) via the scheme implied by Observation 6.7.4 and Observation 6.7.6,
   b) improved upon via Observation 6.7.7 and Observation 6.7.8

Their natural application in information flow control is (backward)-slicing, i.e.: the computation of

[1] non-termination sensitive

$$\left(\left(\rightarrow_{\text{nticd}} \cup \rightarrow_{\text{data}}\right) \ \dot{\cup} \ \rightarrow_{\text{ntiod}}\right)^{*} (M) \tag{7.1}$$

for sets of "observable" nodes M.

For graphs with unique end nodes n_x, this is equivalent to slice along standard control dependence $\rightarrow_{\text{cd}}$ only, i.e.:

$$\left(\rightarrow_{\text{cd}} \cup \rightarrow_{\text{data}}\right)^{*} (M)$$

Here, it usually is of little practical importance whether slicing is performed with regard to only one set M of nodes, or several such sets $M, M', \ldots$: Computation of $\rightarrow_{\text{cd}}$ is — in practice — cheap[2] so in both cases, one just does this, and then slices backwards from $M, M', \ldots$ in the obvious way.

On the other hand, for arbitrary graphs, I must slice as per (7.1). Here, in accordance with the informal slogan

$$\rightarrow_{\text{ntiod}} \quad \approx \quad \left(\sum_{M \in \mathbb{M}} |M|\right) \times \rightarrow_{\text{nticd}}$$

any algorithm computing $\rightarrow_{\text{ntiod}}$ in an explicit representation must be expected to run in $\simeq |N|^2$ steps for typical CFG, and $\mathcal{O}\left(|N|^3\right)$ worst case. To see this directly, consider the n-node looping ladder CFG in Figure 7.1 The variant of this graph which lacks node n and has no edge $(n-1) \rightarrow 0$ is the canonical example of a CFG for which $\rightarrow_{\text{cd}}$ has size $\Theta\left(n^2\right)$. As is, this graph demonstrates that $\rightarrow_{\text{ntiod}}$ can be of size $\Theta\left(n^3\right)$. This is because for every odd node m_1 and every even node m_2, I have

$$n \rightarrow_{\text{ntiod}} (m_1, m_2) \quad \Leftrightarrow \quad n < m_1 \ \wedge \ n \neq m_2$$

[2] In program slicing, $\rightarrow_{\text{cd}}$ often is "mostly" a tree, and hence small, with $|\rightarrow_{\text{cd}}| \simeq |N|$. In fact, if one demands $n_e \rightarrow_G n_x$ and a reducible graphs G, it *is* a tree.

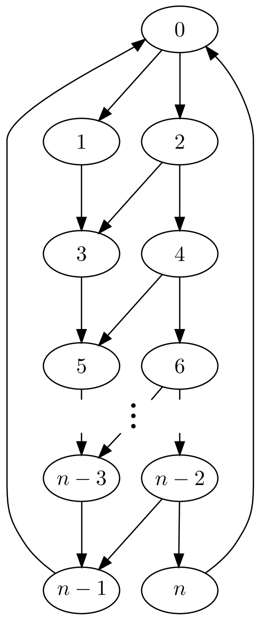


Figure 7.1: The n-node looping ladder CFG.

which means that for every odd m_1, the set of such m_2, n is of size

$$s_{m_1} = \left\lfloor \frac{n}{2} \right\rfloor \cdot \frac{m_1 + 1}{2}$$

and hence:

$$|{\rightarrow}\text{ntiod}| \geq \sum_{m_1 \text{ odd}} s_{m_1} = \left\lfloor \frac{n}{2} \right\rfloor \left(\frac{n-1}{8} + \frac{1}{4} \right) (n-1) \quad \in \quad \Theta\left(n^3\right)$$

If I am interested in the slice (7.1) of only *one* set M of nodes, can I compute this without risking a size $\mathcal{O}\left(|N|^3\right)$ computation? In this chapter, I answer this question in the affirmative.

In Section 7.1 I obtain $\rightarrow_{\text{ntiod}}$, $\rightarrow_{\text{nticd}}$ backwards slice in G by a reduction to *one* computation of $\rightarrow^{G_M}_{\text{nticd}}$ for a suitable subgraph G_M of G. Then, I will show that it is even enough to compute nontermination sensitive postdominance $\sqsupseteq^{G_M}_{\text{SINK}}$ only, and then *directly* read off the $\rightarrow^{G_M}_{\text{nticd}}$-backwards slice of M, without fully computing $\rightarrow^{G_M}_{\text{nticd}}$.

If only the $\rightarrow_{\text{nticd}}$ backwards slice in G is required (e.g.: if all control sinks in the graph G are trivial), I can generalize the technique used in [SG95] to compute the $\rightarrow_{\text{nticd}}$ backwards slice in G of any slicing criterion M. I present this generalization in Appendix C.

In Section 7.2 I present the nontermination sensitive analogue of the result from Section 7.1. In Section 7.3 I present the analogue of the result from Section 7.1 for $\rightarrow_{\text{wod}}$ backward slices. In Section 7.4 I present the analogue of the result from Section 7.1 for *weak control closures*, and then summarize the results in Section 7.5.

7.1 Nontermination Insensitive Slicing

In this section, I will attack the simplified problem of computing the control slice only, i.e.: I disregard data dependencies $\rightarrow_{\text{data}}$. Given a single set M of "observable" nodes in an arbitrary CFG G, the appropriate reduction to *one* computation is remarkably simple. It will turn out that

$$\left(\rightarrow^{G}_{\text{nticd}} \cup \rightarrow^{G}_{\text{ntiod}}\right)^{*}(M) = \left(\rightarrow^{G_{M\not\to}}_{\text{nticd}}\right)^{*}(M) \tag{7.2}$$

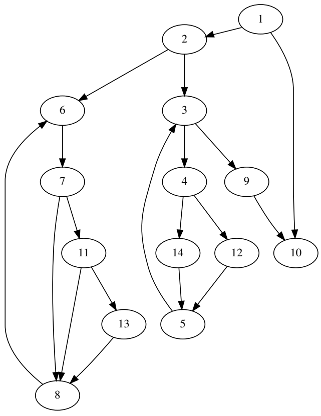
Recall that $G_{M\not\to}$ is the graph obtained from G by removing all edges originating in M.

Intuitively, equation 7.2 holds because by Corollary 6.7.2 on page 102, $n \rightarrow_{\text{ntiod}} (m_1, m_2)$ can only hold for nodes m_1, m_2 that occur in a common cycle C in the pseudo-forest $<_{\text{SINK}}$, and nodes n that either are in the same $<_{\text{SINK}}$ cycle C, or not in any such cycle. By Observation 5.3.5, such cycles C are control sinks in G, i.e., they are strongly connected components of G that have no edge leaving the component. Then by deleting outgoing edges of the slicing criterion M, the component C becomes a region with exit nodes $C \cap M$, and any node n that formerly controlled the order of nodes m_1 and m_2 now controls which of the two nodes is executed *at all*.

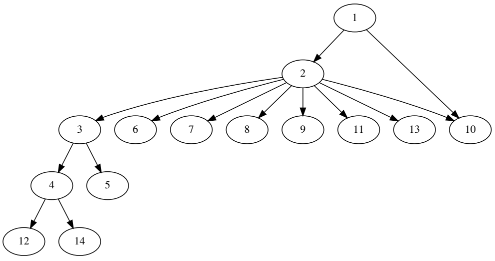
Consider the example from Section 5.3, repeated in Figure 7.2a. The only non-trivial control sink of G is $C = \{6, 7, 8, 11, 13\}$. All nodes in C are nontermination insensitively control dependent on node 2 and 1, but these are the *only* nodes they are control dependent on. For example, node 11 is *not* nontermination insensitively control dependent on node 7. Instead, node $11, 8$ together are nontermination insensitively *order* dependent on node 7: $7 \rightarrow_{\text{ntiod}} (11, 8)$. So consider the slicing criterion $M = \{7, 8\}$, for which the combined backward slice is

$$\left(\rightarrow^{G}_{\text{nticd}} \cup \rightarrow^{G}_{\text{ntiod}}\right)^{*}(M) = \{7, 8, 11, 2, 1\}$$

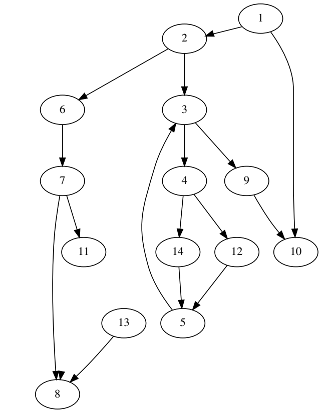
The resulting graph G_M is shown in Figure 7.2c, together with the corresponding nontermination insensitive control dependence in Fig-



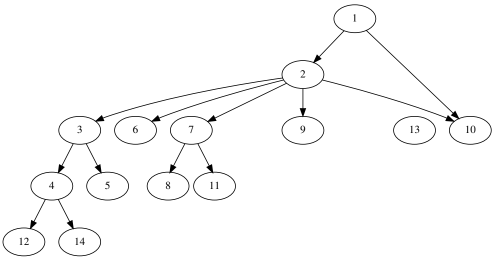
(a) A graph G



(b) $\rightarrow_{\text{nticd}}^{G}$



(c) The Graph G_M



(d) $\rightarrow_{\text{nticd}}^{G_M}$

Figure 7.2: Equation 7.2 for $M = \{8, 11\}$

ure 7.2d. The choice at node $n = 7$ (which formerly decided the *order* of nodes $8, 11$) now controls whether node 8 or node 11 is executed.

Before I can proof equation (7.2), I need a new characterization of the transitive closure of $\rightarrow_{\text{nticd}}$.

Definition 7.1.1. Let G be any CFG, and $m \neq n$ nodes in G. Then $m \notin \text{ipdom}_{\sqsupseteq_{\text{SINK}}}(n)$ is said to be on a path starting in n strictly before any immediate $\sqsupseteq_{\text{SINK}}$-dominator of n — and I write $n \rightarrow_{\text{ntind}} m$ — iff there exists some G-successor x of n such that m is reachable in the graph obtained from G by removing all edges originating in $\text{ipdom}_{\sqsupseteq_{\text{SINK}}}(n)$, i.e.:

$$x \rightarrow^{*}_{G_{\left(\text{ipdom}_{\sqsupseteq_{\text{SINK}}}(n)\right)\not\rightarrow}} m$$

This notion is most easily understood as a generalization of Weiser's transitive notion $\rightarrow_{\text{nd}}$ of control dependence ([Wei81]) for graphs with unique exit node n_x. Recall that (for $n \neq n_x$), $n \rightarrow_{\text{nd}} m$ iff $m \notin \{n, n'\}$ is on a G-path between n and the unique immediate postdominator n' of n. The subscript $_{\text{nd}}$ is derived from the immediate postdominator, which Weiser calls ***n**earest (inverse) **d**ominator*. It is easy to see that $\rightarrow_{\text{ntind}}$ is in fact a generalization of $\rightarrow_{\text{nd}}$, i.e.:

$$n \rightarrow_{\text{ntind}} m \iff n \rightarrow_{\text{nd}} m$$

for graphs with unique exit node n_x. More importantly, it relates to $\rightarrow_{\text{nticd}}$ the same way that $\rightarrow_{\text{nd}}$ does to $\rightarrow_{\text{cd}}$:

Observation 7.1.1. Let G be any CFG, and $m \neq n$ nodes in G. Then

$$n \rightarrow_{\text{ntind}} m \iff n \rightarrow^{*}_{\text{nticd}} m$$

For the proof of (7.2), I also need the following properties of $\rightarrow_{\text{nticd}}$.

Observation 7.1.2. Let G be any CFG, and M any set of nodes in G. Then for all nodes n, m, and all nodes $m_0 \in M$ such that $m \neq m_0$:

$$m \sqsupseteq^{G_{M\not\rightarrow}}_{\text{SINK}} n \implies m \sqsupseteq^{G}_{\text{SINK}[m_0]} n$$

In other words: if m must appear on every $G_{M\not\rightarrow}$-sink-path starting in n, then m must also appear *before* m_0 on every G-sink-path starting in n.

Observation 7.1.3. Let G be any CFG, and M any set of nodes in G. Then for all nodes n, m:

$$m \sqsupseteq^{G_{M \not\to}}_{\text{SINK}} n \quad \Longrightarrow \quad m \sqsupseteq^{G}_{\text{SINK}} n$$

The following fact is well-known, but hitherto only for the special case of graphs with unique exit-node n_x.

Observation 7.1.4. Let G be *any* CFG, and $m_1 \sqsupseteq^{G}_{\text{SINK}} n$ as well as $m_2 \sqsupseteq^{G}_{\text{SINK}} n$. Then, "up to $\sqsupseteq_{\text{SINK}}$-equivalence", the order in which paths starting in n visits m_1, m_2 is fixed, i.e.:

1. $m_1 \sqsupseteq^{G}_{\text{SINK}[m_2]} n$, or
2. $m_2 \sqsupseteq^{G}_{\text{SINK}[m_1]} n$, or
3. both $m_1 \sqsupseteq^{G}_{\text{SINK}} m_2$ and $m_2 \sqsupseteq^{G}_{\text{SINK}} m_1$

Immediate $\sqsupseteq_{\text{SINK}}$-dominators are always visited before non-immediate postdominators:

Observation 7.1.5. Let G be *any* CFG and $n \neq m_2$. Also, let $m_2 \sqsupseteq^{G}_{\text{SINK}} n$ as well as not only $m_1 \sqsupseteq^{G}_{\text{SINK}} n$ but also

$$m_1 \in \text{ipdom}_{\sqsupseteq_{\text{SINK}}}(n)$$

Then, "up to $\sqsupseteq_{\text{SINK}}$-equivalence", any paths starting in n always visits m_1 before m_2, i.e.:

1. $m_1 \sqsupseteq^{G}_{\text{SINK}[m_2]} n$, or
2. both $m_1 \sqsupseteq^{G}_{\text{SINK}} m_2$ and $m_2 \sqsupseteq^{G}_{\text{SINK}} m_1$, as well as

$$m_2 \in \text{ipdom}_{\sqsupseteq_{\text{SINK}}}(n)$$

I need the following three technical observations in the proof of (7.2).

Observation 7.1.6. Let G be any CFG, and M any set of nodes in G. Then whenever

$$m \sqsupseteq^{G}_{\text{SINK}} n \quad \text{but} \quad \neg\, m \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} n$$

there exists some node $m_0 \in M$ and some $G_{M\not\to}$-path π such that $m_0 \neq m$ and

$$\pi = n, \ldots, m_0 \quad \text{such that} \quad m \notin \pi$$

Proof: From $\neg\, m \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} n$ I obtain some $G_{M\not\to}$ sink-path π starting in n with $m \notin \pi$. Then $m_0 \in \pi$ for some $m_0 \in M$, since otherwise π is also a G sink-path with $m \notin \pi$, contradicting $m \sqsupseteq^{G}_{\text{SINK}} n$. With $m_0 \in \pi$, it must be of the form

$$\pi = n, \ldots, m_0$$

□

Observation 7.1.7. Let G be any CFG, and M any set of nodes in G. Then whenever

$$m \sqsupseteq^{G}_{\text{SINK}} n \quad \text{but} \quad \neg\, m \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} n$$

there exists some node $m_0 \in M$ such that

1. $m_0 \neq m$,
2. $m \sqsupseteq^{G}_{\text{SINK}} m_0$,
3. $n \rightarrow^{*}_{G_{M\not\to}} m_0 \rightarrow^{+}_{G} m$, and
4. $\neg\, m \sqsupseteq^{G}_{\text{SINK}[m_0]} n$

Observation 7.1.8. Let G be any CFG, and M any set of nodes in G. Also, let

$$n \left(\rightarrow^{G_{M\not\to}}_{\text{nticd}}\right)^{*} m$$

for some $m \in M$. Then, for all $n' \neq n$:

$$\neg\, n' \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} n$$

Proof (Sketch): By induction on $n \left(\rightarrow^{G_{M\not\to}}_{\text{nticd}}\right)^{*} m$.

Using these observations, I will proof (7.2) via two inductive arguments, starting with the reverse implication.

Theorem 7.1.1. Let G be any CFG, and M any set of nodes in G. Then for any n,

$$\underbrace{n \in \left(\rightarrow^{G}_{\text{nticd}} \ \dot\cup \rightarrow^{\text{G}}_{\text{ntiod}}\right)^{*}(M)}_{=:A} \quad \Leftarrow \quad \underbrace{n \in \left(\rightarrow^{G_{M\not\to}}_{\text{nticd}} \ \dot\cup\ \emptyset\right)^{*}(M)}_{=:B}$$

Proof: By induction on $n \in B$, via Definition 6.2.3. The case $n \in M$ is trivial. For the case $n \rightarrow^{G_{M\not\to}}_{\text{nticd}} n_0$ for some $n_0 \in B$, i can assume $n_0 \in A$. Hence, i also can assume $n \neq n_0$, and also: $n \notin M$, since otherwise I am done. By definition, I have some $G_{M\not\to}$ successors x, x' of n such that

$$\begin{array}{rl} & n_0 \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} x' \quad \text{and hence, by Observation 7.1.3} \quad n_0 \sqsupseteq^{G}_{\text{SINK}} x' \\ \neg & n_0 \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} x \end{array}$$

Of course, the $G_{M\not\to}$-successors of n are exactly the G-successors of n.

If $n \rightarrow^{G}_{\text{nticd}} n_0$ I am done immediately, so I only have to deal with the opposite, i.e.: $\neg\ n \rightarrow^{G}_{\text{nticd}} n_0$. But then, from $n_0 \sqsupseteq^{G}_{\text{SINK}} x'$ i conclude

$$\forall y \text{ s.t. } n \rightarrow_G y.\ n_0 \sqsupseteq^{G}_{\text{SINK}} y \quad \text{and hence} \quad n_0 \sqsupseteq^{G}_{\text{SINK}} n$$

Specifically for x, I have

$$\begin{array}{rrl} & & n_0 \sqsupseteq^{G}_{\text{SINK}} x \\ \text{but:} & \neg & n_0 \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} x \end{array}$$

From Observation 7.1.7, I obtain some $m_0 \in M$ such that

$$\begin{array}{lrl} m_0 \neq n_0 & & n_0 \sqsupseteq^{G}_{\text{SINK}} m_0 \\ x \ \rightarrow^{*}_{G_{M\not\to}} \ m_0 \ \rightarrow^{+}_{G} \ n_0 & \neg & n_0 \sqsupseteq^{G}_{\text{SINK}[m_0]} x \end{array}$$

Now, I discern the two cases $m_0 \sqsupseteq^G_{\text{SINK}} n_0$ and $\neg\, n_0 \sqsupseteq^G_{\text{SINK}} m_0$.

If $m_0 \sqsupseteq^G_{\text{SINK}} n_0$, I intend to show $n \rightarrow_{\text{ntiod}} (n_0, m_0)$ by directly confirming Definition 6.7.1. I already have $n_0 \sqsupseteq^G_{\text{SINK}} n$, and $m_0 \sqsupseteq^G_{\text{SINK}} n$ follows from $m_0 \sqsupseteq^G_{\text{SINK}} n_0$, so I have item (a) covered. For item (b), I want to show $n_0 \sqsupseteq^G_{\text{SINK}[m_0]} x'$, but this follows from $n_0 \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} x'$ and — since $m_0 \neq n_0$ and $m_0 \in M$ – Observation 7.1.2. For item (c), I already have $\neg\, n_0 \sqsupseteq^G_{\text{SINK}[m_0]} x$.

Now if on the other hand, $\neg\, m_0 \sqsupseteq^G_{\text{SINK}} n_0$, I intend to show $n \rightarrow^*_{\text{nticd}} m_0$ in G. Since m_0 in M and only needed to cover the case $n \notin M$, this is equivalent — by Observation 7.1.1 — to showing $n \rightarrow_{\text{ntind}} m_0$ in G. Writing

$$N' = \text{ipdom}_{\sqsupseteq^G_{\text{SINK}}}(n)$$

I will do this by exposing a path

$$\pi = x, \ldots, m_0 \quad \text{such that} \quad \pi \cap N' = \emptyset$$

In order to do this, I differentiate the following cases:

1. $N' = \emptyset$
2. $N' = S$ for some control-sink S in G
3. $N' = \{n'\}$ for some node n'

It is indeed sufficient to cover these cases since $\sqsupseteq^G_{\text{SINK}}$ is the reflexive, transitive closure of some pseudo-forest $>$, and there either n has no $<$-successor, in which case $N' = \emptyset$, or n has a unique $<$-successor n'. Then, if n' lies in some $<$-cycle S, S is a control-sink in G and $N' = S$. Otherwise, $N' = \{n'\}$ (c.f. Observation 5.3.2).

I already have $x \rightarrow^*_{G_{M\not\to}} m_0$ and hence $x \rightarrow^*_G m_0$, so case 1 is trivial.

For case 2, from $n_0 \sqsupseteq^G_{\text{SINK}} n$ I know that $n_0 \in S = N'$, and then from $\neg\, m_0 \sqsupseteq^G_{\text{SINK}} n_0$ I have $m_0 \notin S = N'$ (S is a "root" of $<$). But then for any G-path

$$\pi = x, \ldots, m_0$$

I have $\pi \cap N' = \emptyset$, since i cannot leave a control-sink!

For case 3, I know that $n \neq n'$, since otherwise n is its only $<$-successor, and hence $\neg n'$ 1-$\sqsupseteq_{\text{SINK}}$ n. If $n' = n_0$ then from $\neg\, n_0 \sqsupseteq^G_{\text{SINK}[m_0]}$ x and observing $n_0 \neq m_0$, I obtain a path

$$\pi = x, \ldots, m_0 \quad \text{such that} \quad n_0 \notin \pi$$

which is exactly what I need.

I still need to cover the case $n' \neq n_0$, i.e.: $n_0 \notin N'$. Remembering that I still have $n \neq n_0$, $n_0 \sqsupseteq^G_{\text{SINK}} n$, $n' \in N'$, I use Observation 7.1.5 to conclude

$$n' \sqsupseteq^G_{\text{SINK}[n_0]} n$$

But since $\sqsupseteq^{G_{n_0 \not\to}}_{\text{SINK}}$ is closed under taking a step to $G_{n_0 \not\to}$-successors (i.e.: closed under $\to_{G_{n_0 \not\to}}$) and both $n \neq n_0$ and $n \neq n'$, I have (via two applications of Observation 6.7.5):

$$n' \sqsupseteq^G_{\text{SINK}[n_0]} x'$$

from which I can obtain some path

$$\pi' = x', \ldots, n' \quad \text{such that} \quad n_0 \notin \pi'$$

Again from $\neg\, n_0 \sqsupseteq^G_{\text{SINK}[m_0]} x$ (and still: observing that $m_0 \neq n_0$) I obtain a path

$$\pi = x, \ldots, m_0 \quad \text{such that} \quad n_0 \notin \pi$$

But π cannot contain n', since otherwise, I had a path $\pi'' = n', \ldots, m_0$ with $n_0 \notin \pi''$, and together with π' a path

$$\pi'\ \pi'' = x', \ldots, n', \ldots, m_0 \quad \text{such that} \quad n_0 \notin \pi'\ \pi''$$

in contradiction with the fact that

$$n_0 \sqsupseteq^{G}_{\text{SINK}[m_0]} x'$$

which follows from $n_0 \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} x'$ via Observation 7.1.2 and $m_0 \neq n_0$, $m_0 \in M$. With $n' \notin \pi$, I have concluded case 3 and hence the proof. □

What remains is the forward implication. The proof may — at first — appear to mirror the proof of Theorem 7.1.1. This is because the definition of $\_ \to^{G}_{\text{nticd}} n_0$ requires $n_0 \sqsupseteq_{\text{SINK}} \_$ both in the positive and in the negative for some x, x'. But the actual situations are quite different from the former proof.

Theorem 7.1.2. Let G be any CFG, and M any set of nodes in G. Then for any n,

$$n \in \underbrace{\left(\to^{G}_{\text{nticd}} \cup\!\!\!\cdot\, \to^{G}_{\text{ntiod}}\right)^{*}(M)}_{=:A} \quad \Rightarrow \quad n \in \underbrace{\left(\to^{G_{M\not\to}}_{\text{nticd}} \cup\!\!\!\cdot\, \emptyset\right)^{*}(M)}_{=:B}$$

Proof: By induction on $n \in A$, via Definition 6.2.3. The case $n \in M$ is trivial.

For the case $n \to^{G}_{\text{nticd}} n_0$ for some $n_0 \in A$, I can assume $n_0 \in B$. Hence, I also can assume $n \neq n_0$, and also: $n \notin M$, since otherwise I am done. By definition, I have some G successors x, x' of n such that

$$\begin{array}{ll} & n_0 \sqsupseteq^{G}_{\text{SINK}} x \\ \neg & n_0 \sqsupseteq^{G}_{\text{SINK}} x' \quad \text{and hence, by Observation 7.1.3} \quad \neg n_0 \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} x' \end{array}$$

Since $n \notin M$, the G-successors of n are exactly the $G_{M\not\to}$-successors of n.

If $n \rightarrow_{\text{nticd}}^{G_{M\not\rightarrow}} n_0$ I am done immediately, so I only have to deal with the opposite, i.e.: $\neg\ n \rightarrow_{\text{nticd}}^{G_{M\not\rightarrow}} n_0$. But then, from $\neg\ n_0 \sqsupseteq_{\text{SINK}}^{G_{M\not\rightarrow}} x'$ I conclude

$$\forall y \text{ s.t. } n \rightarrow_G y.\ \neg\ n_0 \sqsupseteq_{\text{SINK}}^{G_{M\not\rightarrow}} y$$

Specifically for x, I have

$$\begin{array}{rll} & & n_0 \sqsupseteq_{\text{SINK}}^{G} x \\ \text{but:} & \neg & n_0 \sqsupseteq_{\text{SINK}}^{G_{M\not\rightarrow}} x \end{array}$$

From Observation 7.1.6, I obtain some $m_0 \in M$ with $m_0 \neq n_0$ and a $G_{M\not\rightarrow}$-path

$$\pi = x, \ldots, m_0 \quad \text{such that} \quad n_0 \notin \pi$$

I will show that there exists no $n' \neq n$ such that $n' \sqsupseteq_{\text{SINK}}^{G_{M\not\rightarrow}} n$. Once I have done that, I can conclude that

$$N' = \text{ipdom}_{\sqsupseteq_{\text{SINK}}^{G_{M\not\rightarrow}}}(n) = \emptyset$$

and immediately get $n \rightarrow_{\text{nticd}}^{*} m_0$ in $G_{M\not\rightarrow}$ from $n \rightarrow_{\text{ntind}} m_0$ (in $G_{M\not\rightarrow}$), which then holds because of $x \rightarrow_{G_{M\not\rightarrow}}^{*} m_0$.

In order to show that there exists no $n' \neq n$ such that $n' \sqsupseteq_{\text{SINK}}^{G_{M\not\rightarrow}} n$, I assume the opposite. Then also $n' \sqsupseteq_{\text{SINK}}^{G_{M\not\rightarrow}} x$ and $n' \sqsupseteq_{\text{SINK}}^{G_{M\not\rightarrow}} x'$. I also obtain from $\neg\ n_0 \sqsupseteq_{\text{SINK}}^{G} x'$ some G-path

$$\pi' = x', \ldots\ldots\ldots\ldots\ldots\ldots, z \quad \text{such that} \quad n_0 \notin \pi' \text{ and } \neg\ z \rightarrow_G^{*} n_0$$

I discern the two cases $\exists m' \in M.\ m' \in \pi'$ and its negation, both of which I have to lead to a contradiction. If there exists such a m', I can choose m' as the one appearing first in π'. Then from $n' \sqsupseteq_{\text{SINK}}^{G_{M\not\rightarrow}} n$ I conclude that n' must appear in π' before (or at) this occurrence. In summary, I have[3]

$$\pi' = x', \ldots, \underbrace{n', \ldots, m', \ldots, z}_{=:\pi'_0} \quad \text{with} \quad n_0 \notin \pi' \text{ and } \neg\, z \to^*_G n_0$$

Now, for the $G_{M\not\to}$-path

$$\pi = x, \ldots, m_0 \quad \text{such that} \quad n_0 \notin \pi$$

I infer from $n' \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} n$ that $n' \in \pi$, and hence have a prefix $\pi_0 = x, \ldots, n'$ with $n_0 \notin \pi$. But then I have

$$n_0 \;\notin\; \pi_0\pi'_0 \;=\; x, \ldots, n', \ldots, m', \ldots, z \quad \text{with} \neg\, z \to^*_G n_0$$

in contradiction to $n_0 \sqsupseteq^{G}_{\text{SINK}} x$.

In order derive a contradiction from $\neg\, \exists m' \in M.\ m' \in \pi'$, I observe that then π' is not only a G-path but also a $G_{M\not\to}$-path. This means that either $n' \in \pi'$, i.e.:

$$\pi' = x', \ldots, \underbrace{n', \ldots, z}_{=:\pi'_0} \quad \text{such that} \quad n_0 \notin \pi' \text{ and } \neg\, z \to^*_G n_0$$

in which case I obtain a contradiction just as I did before, *or* there exists a continuation π'' of π' in $G_{M\not\to}$ such that $n' \in \pi''$, i.e.:

$$\pi'\pi'' = x', \ldots, z,\ \ldots, n'$$

But since $\neg\, z \to^*_G n_0$, I have $n_0 \notin \pi''$ and $\neg\, n' \to^*_G n_0$. Choosing $\pi_0 = x, \ldots, n'$ with $n_0 \notin \pi_0$ as before, this contradicts $n_0 \sqsupseteq^{G}_{\text{SINK}} x$. This concludes the inductive case $n \to^{G}_{\text{nticd}} n_0$.

[3] not demanding x', n', m', z to be mutually different

What is left is the inductive case $n \rightarrow^{G}_{\text{ntiod}} (m_1, m_2)$ for some $m_1, m_2 \in A$, in which case I can assume $m_1, m_2 \in B$. Hence, I also can assume $n \neq m_1$, $n \neq m_2$ and also: $n \notin M$, since otherwise I am done.

I have $m_1 \neq m_2$ by definition of $\rightarrow_{\text{ntiod}}$, so by Observation 7.1.8 and $m_2 \in B$ I conclude

$$\neg\ m_1 \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} m_2$$

Also by definition, I have some G successors x, x' of n such that

$$\begin{array}{rl} & m_1 \sqsupseteq^{G}_{\text{SINK}[m_2]} x \\ \neg & m_1 \sqsupseteq^{G}_{\text{SINK}[m_2]} x' \end{array}$$

as well as

$$\begin{array}{rlcl} & m_1 \sqsupseteq^{G}_{\text{SINK}} n & \text{and since } n \neq m_1 & m_1 \sqsupseteq^{G}_{\text{SINK}} x' \\ \neg & m_2 \sqsupseteq^{G}_{\text{SINK}} n & & \end{array}$$

Since $n \notin M$, the G-successors of n are exactly the $G_{M\not\to}$-successors of n.

From $\neg\ m_1 \sqsupseteq^{G}_{\text{SINK}[m_2]} x'$ I obtain a G-path

$$\pi' = x', \ldots, m_2 \quad \text{such that} \quad m_1 \notin \pi'$$

If $n \rightarrow^{G_{M\not\to}}_{\text{nticd}} m_1$ I am done immediately, so I only have to deal with the opposite, i.e.: $\neg\ n \rightarrow^{G_{M\not\to}}_{\text{nticd}} m_1$. So either

$$\forall y \text{ s.t. } n \rightarrow_G y.\ \neg\ m_1 \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} y$$

or $\forall y$ s.t. $n \rightarrow_G y.\ m_1 \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} y$. But the latter cannot be, since if it did hold, I would have $m_1 \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} x'$ from which it would follow — since $m_1 \notin \pi'$ — that π' is not only a G-path, but also a $G_{M\not\to}$-path, and also that any $G_{M\not\to}$-sink-path starting in m_2 needed to contain m_1, contradicting $\neg\ m_1 \sqsupseteq^{G_{M\not\to}}_{\text{SINK}} m_2$.

I will show that there exists no $n' \neq n$ such that $n' \sqsupseteq_{\text{SINK}}^{G_{M\not\to}} n$. Once I have done that, I can conclude that

$$N' = \text{ipdom}_{\sqsupseteq_{\text{SINK}}^{G_{M\not\to}}}(n) = \emptyset$$

and immediately get $n \to_{\text{nticd}}^{*} m_0$ in $G_{M\not\to}$ from $n \to_{\text{ntind}} m_0$ (in $G_{M\not\to}$) for some $m_0 \in M$, which must exist because of

$$\begin{aligned} m_1 \sqsupseteq_{\text{SINK}}^{G} x &\quad\Longrightarrow\quad x \to_{G}^{*} m_1 \\ m_1 \in B &\quad\Longrightarrow\quad \exists m_0' \in M. m_1 \to_{G}^{*} m_0' \end{aligned}$$

but any G-path $x, \ldots, m_0'$ must either already be a $G_{M\not\to}$-path, or interrupted by some $m_0 \in M$.

In order to show that there exists no $n' \neq n$ such that $n' \sqsupseteq_{\text{SINK}}^{G_{M\not\to}} n$, I assume the opposite. Then also $n' \sqsupseteq_{\text{SINK}}^{G_{M\not\to}} x$ and $n' \sqsupseteq_{\text{SINK}}^{G_{M\not\to}} x'$. I discern the two cases $\exists m' \in M.\ m' \in \pi'$ and its negation, both of which I have to lead to a contradiction. If there exists such a m', I can choose chose m' as the one appearing first in π'. Then from $n' \sqsupseteq_{\text{SINK}}^{G_{M\not\to}} n$ I conclude that n' must appear in π' before (or at) this occurrence. In summary, I have[4]

$$\pi' = x', \ldots, \underbrace{n', \ldots, m', \ldots, m_2}_{=:\pi_0'} \quad \text{with} \quad m_1 \notin \pi'$$

Now, from $\neg\, m_1 \sqsupseteq_{\text{SINK}}^{G_{M\not\to}} x$ I obtain some $m \in M$ and some $G_{M\not\to}$-path

$$\pi = x, \ldots, m \quad \text{such that} \quad m_1 \notin \pi$$

and infer from $n' \sqsupseteq_{\text{SINK}}^{G_{M\not\to}} n$ that $n' \in \pi$, and hence have a prefix $\pi_0 = x, \ldots, n'$ with $m_1 \notin \pi$. But then I have

$$m_1 \;\notin\; \pi_0\pi_0' \;=\; x, \ldots, n', \ldots, m', \ldots, m_2$$

[4] not demanding x', n', m', m_2 to be mutually different

in contradiction to $m_1 \sqsupseteq^{G}_{\mathrm{SINK}[m_2]} x$.

In order derive a contradiction from $\neg\ \exists m' \in M.\ m' \in \pi'$, I observe that then π' is not only a G-path but also a $G_{M\not\to}$-path, and $m_2 \notin M$. If $n' \in \pi'$, I derive a contradiction as before. Otherwise, I have $n' \neq m_2$, and because of $n' \sqsupseteq^{G_{M\not\to}}_{\mathrm{SINK}} x'$, any $G_{M\not\to}$-sink-path continuing π'[5] must contain n', i.e.: $n' \sqsupseteq^{G_{M\not\to}}_{\mathrm{SINK}} m_2$. But by Observation 7.1.8 and $m_2 \in B$ this contradicts $n' \neq m_2$. □

This concludes the proof of Equation 7.2, but I am not quit finished: in many applications, I will need to slice w.r.t to some additional binary relation $\to_{\mathrm{d}}$ such as, e.g., data-dependence $\to_{\mathrm{data}}$. I do not give formal proof but merely observe that under just one weak condition, this is possible as well:

Observation 7.1.9. Let G be any CFG, M any set of nodes, and $\to_{\mathrm{d}}$ be any binary relation on nodes compatible with $\to_G$, i.e.:

$$n \to_{\mathrm{d}} m \quad\Longrightarrow\quad n \to_G^* m$$

Then

$$\left(\to^{G}_{\mathrm{nticd}} \cup \to_{\mathrm{d}}\ \dot\cup \to^{G}_{\mathrm{ntiod}}\right)^* (M) = \left(\to^{G_{M\not\to}}_{\mathrm{nticd}} \cup \to_{\mathrm{d}}\right)^* (M)$$

7.1.1 A Direct Algorithm for Control Slicing

Given Equation 7.2, i.e.:

$$\left(\to^{G}_{\mathrm{nticd}}\ \dot\cup \to^{G}_{\mathrm{ntiod}}\right)^* (M) = \left(\to^{G_{M\not\to}}_{\mathrm{nticd}}\right)^* (M)$$

I can obviously compute the non-termination insensitive backward control slice of M using only $\to^{G_{M\not\to}}_{\mathrm{nticd}}$, without the need to compute

[5] i.e.: starting in some $G_{M\not\to}$-successor of m_2

$\rightarrow$ntiod. But I do not even need $\rightarrow^{G_{M\not\rightarrow}}_{\text{nticd}}$. Thanks to the fact that in $G_{M\not\rightarrow}$, nodes n (transitively) controlling nodes $m \in M$ have no successor in the corresponding pseudo-forest $<^{G_{M\not\rightarrow}}_{\text{SINK}}$, and thanks to the characterization of such n via $\rightarrow^{G_{M\not\rightarrow}}_{\text{ntind}}$, it is enough to compute $<^{G_{M\not\rightarrow}}_{\text{SINK}}$, and then enumerate those n following a reverse depth search in $G_{M\not\rightarrow}$ starting in M:

Lemma 7.1.1.

$$\left(\rightarrow^{G_{M\not\rightarrow}}_{\text{nticd}}\right)^{*}(M) = \{\, n \mid n \rightarrow^{*}_{G_{M\not\rightarrow}} m,\ m \in M,\ \neg n <^{G_{M\not\rightarrow}}_{\text{SINK}} \textvisiblespace \,\}$$

Proof: Directly from Observation 7.1.8 and Observation 7.1.1. □

7.2 Nontermination Sensitive Slicing

Given the reduction to $(\to_{\text{nticd}})^*$ from equation (7.2), i.e.:

$$\left(\to^G_{\text{nticd}} \ \dot{\cup} \to^G_{\text{ntiod}}\right)^* (M) = \left(\to^{G_{M \not\to}}_{\text{nticd}}\right)^* (M)$$

it is natural to ask whether slices w.r.t. other ternary relations can be obtained similarly. For $\to_{\text{ntsod}}$ this is indeed the case:

Observation 7.2.1. Let G be any CFG, and M any set of nodes in G. Then

$$\left(\to^G_{\text{ntscd}} \ \dot{\cup} \to^G_{\text{ntsod}}\right)^* (M) = \left(\to^{G_{M \not\to}}_{\text{ntscd}}\right)^* (M)$$

I do not give proof, but merely note that the analogue of the characterization Observation 7.1.1 of the transitive closure of $\to_{\text{nticd}}$ does indeed hold, simply by replacing $\sqsupseteq_{\text{SINK}}$ with $\sqsupseteq_{\text{MAX}}$.

Definition 7.2.1. Let G be any CFG, and $m \neq n$ nodes in G. Then $m \notin \text{ipdom}_{\sqsupseteq_{\text{MAX}}}(n)$ is said to be on a path starting in n strictly before any immediate $\sqsupseteq_{\text{MAX}}$-dominator of n — and I write $n \to_{\text{ntsnd}} m$ — iff

$$x \to^*_{G_{\left(\text{ipdom}_{\sqsupseteq_{\text{MAX}}}(n)\right) \not\to}} m$$

Observation 7.2.2. Let G be any CFG, and $m \neq n$ nodes in G. Then

$$n \to_{\text{ntsnd}} m \quad \iff \quad n \to^*_{\text{ntscd}} m$$

Also, i do not need to compute $\to^{G_{M \not\to}}_{\text{ntscd}}$, but it is enough to compute $<^{G_{M \not\to}}_{\text{MAX}}$.

Observation 7.2.3. Let G be any CFG, and M any set of nodes in G. Also, let

$$n \left(\to^{G_{M \not\to}}_{\text{ntscd}}\right)^* m$$

for some $m \in M$. Then, for all $n' \neq n$:

$$\neg\, n' \sqsupseteq^{G_{M\not\to}}_{\text{MAX}} n$$

Lemma 7.2.1.

$$\left(\to^{G_{M\not\to}}_{\text{ntscd}}\right)^* (M) = \{\, n \mid n \to^*_{G_{M\not\to}} m,\ m \in M, \neg n <^{G_{M\not\to}}_{\text{MAX}} \textvisiblespace \,\}$$

Proof: Directly from Observation 7.2.3 and Observation 7.2.2. □

Since ultimately, I am interested in slices respecting not only control- but also data-dependencies, I also need the following observation:

Observation 7.2.4. Let G be any CFG, M any set of nodes, and $\to_{\text{d}}$ be any binary relation on nodes compatible with $\to_G$, i.e.:

$$n \to_{\text{d}} m \quad \Longrightarrow \quad n \to^*_G m$$

Then

$$\left(\to^{G}_{\text{ntscd}} \cup \to_{\text{d}} \ \dot\cup \to^{\text{G}}_{\text{ntsod}}\right)^* (M) = \left(\to^{G_{M\not\to}}_{\text{ntscd}} \cup \to_{\text{d}}\right)^* (M)$$

7.3 Weak Order Control Slices

Recall from the discussion of Figure 6.4 on page 87 that slices w.r.t $\rightarrow_{\text{nticd}}$, $\rightarrow_{\text{ntiod}}$ are larger than slices w.r.t $\rightarrow_{\text{wod}}$ because the former demand that the slice includes any node n *which can make another node in the slice unreachable*. This in mind, and given the success of the reduction to $G_{M\not\rightarrow}$ in equation (7.2), my following reduction is no longer surprising:

Observation 7.3.1. Let G be any CFG, and M any set of nodes in G, and let

$$G_M = G^{\rightarrow^* M}_{M\not\rightarrow}$$

be the CFG obtained from G by removing any nodes that cannot reach M, and then any outgoing edges of M. Then

$$\left(\rightarrow^{\text{G}}_{\text{wod}}\right)^* (M) = \left(\rightarrow^{G_M}_{\text{nticd}}\right)^* (M)$$

In algorithmic terms, Observation 7.3.1 says that for a given query $\left(\rightarrow^{\text{G}}_{\text{wod}}\right)^* (M)$, I can compute $<^{G_M}_{\text{SINK}}$ and then (by Algorithm 17 or Lemma 7.1.1) $\left(\rightarrow^{G_M}_{\text{nticd}}\right)^* (M)$.

Remark 7.3.1. When faced with multiple queries $M_1, M_2, \ldots$, for the same CFG G, it is in fact possible to compute $<^{G_M}_{\text{SINK}}$ from $<^{G}_{\text{SINK}}$ by an incremental algorithm vaguely similarly to the computation of $<_{m_2'}$ from $<_{m_2}$ in Observation 6.7.8 on page 107, but I do not describe it here. Note that in the extreme case, where G consists of a single sink N, $<^{G}_{\text{SINK}}$ is completely useless for the computation of $<^{G_M}_{\text{SINK}}$ since then $<^{G}_{\text{SINK}}$ is just one big cycle containing every node N, and carries no information about the control structure of G at all.

7.4 Weak Control Closures

Recall (Definition 6.5.2) that a set $S \subseteq N$ was defined to be weakly control closed in G iff all vertices $n \notin S$ reachable from S are S-weakly committing (i.e.: $\mathrm{obs}_S^G(n)$ is at most a singleton) in G. In [Dan+11], the same authors also define the notion of *weakly deciding* nodes:

Definition 7.4.1 (adapting [Dan+11], Definition 50)**.** Given a CFG $G = (N, E)$ and a set $M \subseteq N$, a node $n \notin M$ is M-weakly deciding in G if there exist paths

$$\begin{aligned} \pi_l &= \underbrace{n, \ldots,}_{\notin S} m_l \quad \text{such that} \quad m_l \in M \\ \pi_r &= \underbrace{n, \ldots,}_{\notin S} m_r \quad \text{such that} \quad m_r \in M \end{aligned}$$

with $\pi_l \cap \pi_r = \{n\}$. Specifically: $m_l \neq m_r$.

Definition 7.4.2. I denote with $\mathrm{WD}_G(M)$ the set of M-weakly deciding nodes in G, and with $\mathrm{WCC}_G(M)$ the smallest set $S \supseteq M$ that is weakly control closed in G.

Lemma 7.4.1 ([LKL18], Property 4)**.** Recall that for a CFG $G = (N, E)$ and $M \subseteq N$, $N^{M\rightarrow^*}$ is the set of nodes reachable from M in G.

$$\mathrm{WCC}_G(M) = (M \cup \mathrm{WD}_G(M)) \ \cap \ N^{M\rightarrow^*}$$

Weak control closures and the set of weakly deciding nodes relate to weak order dependence and nontermination-insentitive control dependence as follows:

Observation 7.4.1.

$$(\rightarrow_{\mathrm{wod}}^{\mathrm{G}})^* (M) = M \cup \mathrm{WD}_G(M)$$

Also

$$\begin{array}{rclccl}
\mathrm{WCC}_G(M) & = & (\rightarrow^{G}_{\mathrm{wod}})^*(M) & \cap & N^{M\rightarrow^*} & \\
 & = & (\rightarrow^{G_M}_{\mathrm{wod}})^*(M) & & & \text{for} \quad G_M = G^{M\rightarrow^*} \\
 & = & \left(\rightarrow^{G_M}_{\mathrm{nticd}}\right)^*(M) & \cap & N^{M\rightarrow^*} & \text{for} \quad G_M = G^{\rightarrow^* M}_{M\not\rightarrow} \\
 & = & \left(\rightarrow^{G_M}_{\mathrm{nticd}}\right)^*(M) & & & \text{for} \quad G_M = G^{M\rightarrow^* M}_{M\not\rightarrow}
\end{array}$$

7.5 The Role of $\rightarrow_{\text{nticd}}$ for Nontermination Insensitive Slices

In review of this chapter, I want to comment on the role of nontermination insensitive control dependence $\rightarrow_{\text{nticd}}$, originally defined in [Ran+07].

While it was meant to generalize standard control dependency $\rightarrow_{\text{cd}}$, the fact that $\rightarrow_{\text{nticd}}$ does not capture any dependencies *within* control-sinks begs the question: What *is* the purpose of $\rightarrow_{\text{nticd}}$?

In fact in [Amt08], $\rightarrow_{\text{nticd}}$ was abandoned in favor of $\rightarrow_{\text{wod}}$ by its inventors. But in Chapter 6 I showed that "with a little help from" nontermination insensitive order dependence $\rightarrow_{\text{ntiod}}$, nontermination insensitive control dependence $\rightarrow_{\text{nticd}}$ is indeed a sound (and minimal!) basis for nontermination insensitive slicing.

Even more, by moving from G to an appropriate CFG G_M, $\rightarrow^{G_M}_{\text{nticd}}$ *alone* can in fact be used to obtain what I think are the three relevant notions of nontermination insensitive slicing of arbitrary CFG:

$$\begin{aligned}
\left(\rightarrow^{G}_{\text{nticd}} \mathbin{\dot\cup} \rightarrow^{\text{G}}_{\text{ntiod}}\right)^* (M) &= \left(\rightarrow^{G_M}_{\text{nticd}}\right)^* (M) \ \text{ for } G_M = G_{M\not\rightarrow} \\
\left(\rightarrow^{\text{G}}_{\text{wod}}\right)^* (M) &= \left(\rightarrow^{G_M}_{\text{nticd}}\right)^* (M) \ \text{ for } G_M = G^{\rightarrow^* M}_{M\not\rightarrow} \\
\text{WCC}_G (M) &= \left(\rightarrow^{G_M}_{\text{nticd}}\right)^* (M) \ \text{ for } G_M = G^{M\rightarrow^* M}_{M\not\rightarrow}
\end{aligned}$$

Due to the fact that all three G_M are of the form $G'_{M\not\rightarrow}$, Lemma 7.1.1 is applicable and I do *not* need to explicitly compute $\rightarrow^{G_M}_{\text{nticd}}$. The only requirement is the corresponding nontermination insensitive postdominance pseudo-forest $<^{G_M}_{\text{SINK}}$.

I stress that in this application, the crucial step in the generalization from postdominance trees $<_{\text{POST}}$ for CFG with unique exit node n_x, to pseudo-forests $<_{\text{SINK}}$ for arbitrary CFG is not the *pseudo*-part (i.e.:

the presence of cycles in $<_{\text{SINK}}$), but the *forest*-part: The transformation $G \mapsto G'_{M \not\to}$ only ever introduce proper *trees* (among them: those with roots M). "Pseudoness" is not important here —in fact, the second and third variant of G_M lead to proper forests, with the each $m \in M$ being a root.

Originally, I specifically designed Algorithm 17 from Appendix C to compute slices in G_M, and it does this adequately. Computation via Lemma 7.1.1, however, is much simpler, and I probably would not have bothered with design of Algorithm 17 but for the fact that at that time, I had not yet discovered Lemma 7.1.1, yet! As is, Algorithm 17 retains useful for the case of multiple queries $M_1, M_2, \ldots$ for CFG G without non-trivial sinks.

Summary

- Nontermination insensitive order dependence $\to_{\text{ntiod}}$ can be of size $\mathcal{O}\left(|N|^3\right)$.
- But nontermination insensitive slices can be computed directly from nontermination insensitive postdominance $\sqsupseteq_{\text{SINK}}$, in a suitable graph G_M obtained from the slicing criterion M.

8 Performance Benchmarks

Atticus told me to delete the adjectives and I'd have the facts.

(Harper Lee — To Kill a Mockingbird)

I evaluated the performance of all new algorithms from Part I of this thesis on three classes of graphs:

1. Control flow graphs of Java methods, as generated by the JOANA system for various third party Java programs
2. Randomly generated graphs $G = (N, E)$ usually with $|E| = 2|E|$, as generated by the standard generator from the JGraphT[NP19] library
3. Variants of "ladder" graphs (see Figure 7.1 on page 119), intended to expose "Bad Case" behavior.

Except for the benchmarks concerned with nontermination insensitive order dependence $\rightarrow_{\text{ntiod}}$, the ladder graphs I use will be unique-exit-node ladder graphs. This sometimes allows me to directly compare with existing algorithms for such graphs.

All benchmarks in this chapter were made on a "desktop workstation" class computer with an Intel i7-6700 CPU at 3.40GHz, and 64 gigabyte RAM. I implemented the algorithms in the Java programming language, and used the OpenJDK Java 9 VM to run them. All benchmarks were run using the Java Microbenchmark Harness JMH[Cor20].

Unless explicitly stated otherwise, all data points represent the average over $n + 1$ runs of the benchmark, where n is at least the number of runs which can be finished within 1 second. For example, the data point at $|N| = 21076$, time $=$ 18ms in Figure 8.1a stands for the average of at least ≈ 50 runs of the benchmark that finished within 1 second. On the other hand, the data point in at $|N| = 65000$, time $=$ 88s in Figure 8.1c results from only one run of the benchmark.

The purpose of these benchmarks is to give a general idea of the scalability of the algorithms. For example, the benchmark in the upper left and upper right of Figure 8.3 suggest that my new algorithm for the computation of nontermination sensitive control dependence $\rightarrow_{\text{ntscd}}$ appears to scale “almost linearly” for “average” input CFG, while the original algorithm from [Ran+07] is clearly grows super-linearly for such graphs.

The central outcome of the benchmarks in this chapter is:

1. For “average” CFG, all my algorithms for control dependence variants for arbitrary graphs offer performance “almost linear” in the size of the graph.
2. But for “bad case” CFG, some algorithms perform for decidedly super-linear, and become impractical for very large such graphs.

I also include benchmarks for *timing sensitive* control dependence, *timing* dependence, and *timing sensitive* postdominance, as to be introduced later in part II of this thesis. There the central outcome is:

1. Timing sensitive postdominance can be computed in “almost linear” time for “average” CFG.
2. But even for “average” CFG, timing sensitive control dependence and timing dependence scale super-linearly.

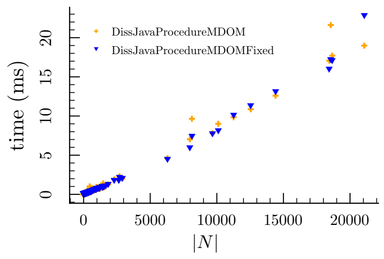
8.1 Nontermination Sensitive Postdominance

In Subsection 5.2.1, I introduced Algorithm 5 for the computation of maximal path postdominance $\sqsupseteq_{\text{MAX}}$, represented as a pseudo-forest $<_{\text{MAX}}$. This algorithm requires the computation of least common ancestors $\text{lca}_<$ in pseudo-forests $<$. I introduced two variants thereof (Algorithm 3 on page 47, and Algorithm 18 on page 396), of which I use the latter.

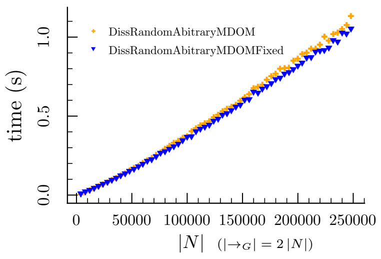
The Algorithm 5 implements chaotic iteration, by reinserting into a workset those nodes affected by modification to the pseudo-forest. In contrast, the Algorithm 19 on page 397 repeatedly iterates in a fixed node order.

Both these variants do not specify an iteration order (i.e.: Algorithm 5 does not specify which node x to remove from the work set at the start of each iteration, and Algorithm 19 does not specify the initial order of nodes in the workqueue). By default, my implementations orders the nodes reversed-topologically (as computed by an implementation of Kosaraju's Algorithm for strongly connected components, with nodes in the same strongly connected component ordered arbitrarily). For Algorithm 5 this means that at the beginning of each iteration, the *rightmost* node (by topological ordering) is removed from the workset.

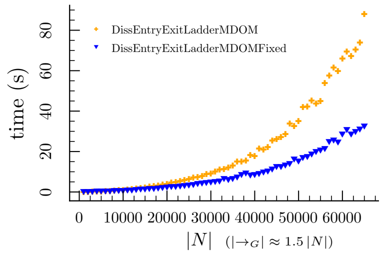
For Java CFG and randomly generated graphs (neither necessarily with unique exit node), Algorithm 5 (✚) and Algorithm 19 (▼) behave similarly (Figure 8.1a and Figure 8.1b). Ladder graphs expose non-linear *bad-case* behavior (Figure 8.1c). This is even more pronounced when additionally, I deliberately choose a bad iteration order (Figure 8.1d).



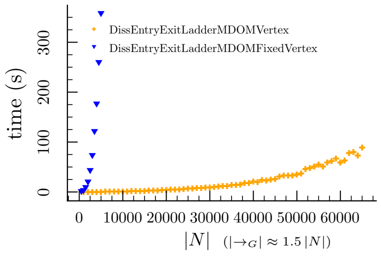
(a) Java CFG



(b) Random Graphs



(c) “Bad Case”



(d) “Bad Case”, bad iteration order

Figure 8.1: Computation of $<_{\text{MAX}}$.

8.2 Nontermination Insensitive Postdominance

In Section 5.3, I introduced Algorithm 6 for the computation of sink path postdominance $\sqsupseteq_{\text{SINK}}$, represented as a pseudo-forest $<_{\text{SINK}}$. Just as before, it uses Algorithm 18 for the computation of least common ancestors $\text{lca}_{<}$.

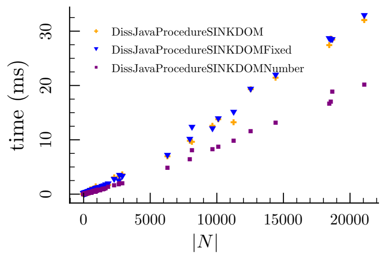
Again, Algorithm 6 implements chaotic iteration, by reinserting into a workset those nodes affected by modification to the pseudo-forest. I also implemented a variant of Algorithm 6 in which the downward fixed point phase repeatedly iterates a workqueue of nodes in a fixed node order. Again, neither variant specifies an iteration order. As before, implementations by defaults orders the nodes reversed-topologically. Unlike before, this ordering does not require an additional step, since the strongly connected component computation it can be obtained from is necessary anyway, in order to find *control sinks*, which are exactly those components without outgoing edges.

In Section D.2 in the appendix, I introduce Algorithm 24 for the computation of $\sqsupseteq_{\text{SINK}}$. This algorithm computes least common ancestors $\text{lca}_{<}$ by comparison of postorder numbers, via Algorithm 23. By default, nodes are iterated in reversed-topological order.

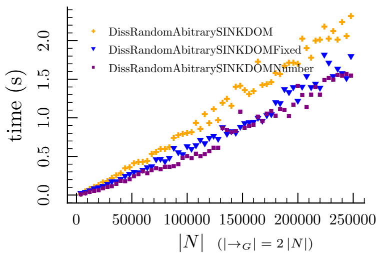
For Java CFG (Figure 8.2a) the fixed-iteration order variant of Algorithm 6 (▼) performs on par with the original Algorithm 6 (✚). For randomly generated graphs (Figure 8.2b)[1] the variant (▼) appears to perform a bit better than the original (✚) for very large graphs, roughly on-par with Algorithm 24 (■).

Using reversed-topological iteration order, ladder graphs (Figure 8.2c) expose non-linear *bad-case* behavior only for Algorithm 6 (✚) and its variant (▼). Even with a deliberately bad iteration order, I could not produce much worse performance for these two algorithm (Figure 8.2d). Presumably, the iteration in upward phase SINK_{up} (which is the same both variants) finds a good-enough approximation. On

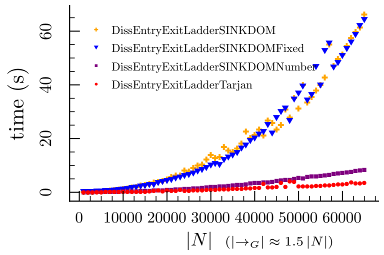
[1] just as the Java CFG, not necessarily with unique exit node



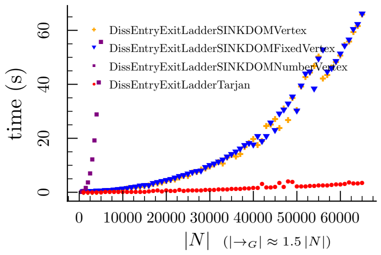
(a) Java CFG



(b) Random Graphs



(c) "Bad Case"



(d) "Bad Case", bad iteration order

Figure 8.2: Computation of $<_{\text{SINK}}$.

the other hand, Algorithm 24 (■) is affected heavily by a deliberately bad iteration order.

The ladder graphs I use are unique-exit-node ladder graphs, since if I used the ladder graph from Figure 7.1 on page 119 *as shown*, the whole graph would form one big control-sink, and $<_{\text{SINK}}$ one big cycle, making the performance comparison moot. This also allows me to directly compare with an implementation of the algorithm by Lengauer, Tarjan [LT79] (●).

8.3 Generalized Postdominance Frontiers

In Section 3.2, I introduced Algorithm 1 for the computation generalized postdominance frontiers.

Nontermination Sensitive Control Dependence When instantiated with $<_{\text{MAX}}$, this yields an algorithm for nontermination sensitive control dependence $\rightarrow_{\text{ntscd}}$. The benchmarks include the computation time of both Algorithm 1, and $<_{\text{MAX}}$ via Algorithm 19 (▼). I compare with Algorithm 14 from [Ran+07] as shown on page 363 (✚).

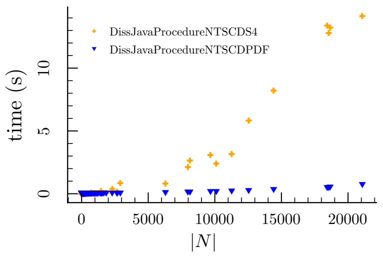
For Java CFG and randomly generated graphs, Algorithm 14 becomes impractical for moderately sized graphs, while Algorithm 1 performs well even for very large graphs (Figure 8.3, upper left and right).

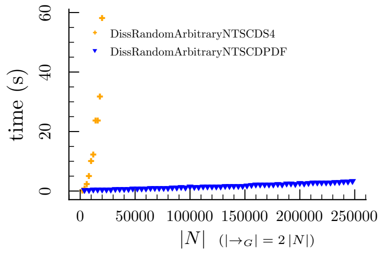
Ladder graphs expose non-linear *bad-case* behavior even for Algorithm 1 (Figure 8.3c). This cannot be circumvented, since in these ladder graphs, the size of the relation $\rightarrow_{\text{nticd}}$ is quadratic in the number of nodes (similar to the size of relation $\rightarrow_{\text{ntiod}}$ being cubic for the ladder graphs on page 119).

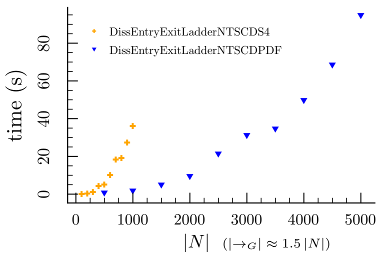
Nontermination Insensitive Control Dependence When instantiated with $<_{\text{SINK}}$, Algorithm 1 yields an algorithm for nontermination sensitive control dependence $\rightarrow_{\text{ntscd}}$. The benchmarks include the computation time of both Algorithm 1, and $\sqsupseteq_{\text{SINK}}$ via the variant of Algorithm 19 with fixed iteration order (▼). Since the Algorithm 15 from [Ran+07] is incorrect, I do not compare with it. Instead, I compare with Algorithm 16 as shown on page 380 (✚).

For Java CFG and randomly generated graphs, Algorithm 16 becomes impractical for moderately sized graphs, while Algorithm 1 performs well even for very large graphs (Figure 8.4, upper left and right).

Ladder graphs expose non-linear *bad-case* behavior even for Algorithm 1 (Figure 8.4c), which again cannot be circumvented.

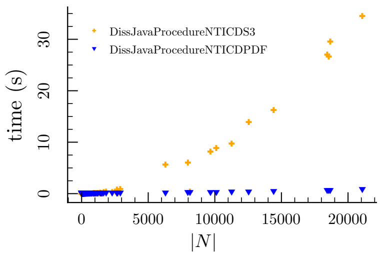


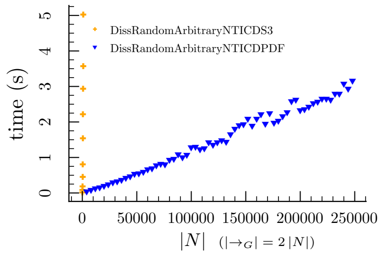


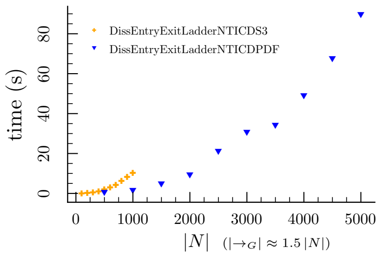


(c) "Bad Case"

Figure 8.3: Computation of $\rightarrow$ntscd.







(c) "Bad Case"

Figure 8.4: Computation of $\rightarrow$nticd.

8.4 Control Slices

Given $\rightarrow_{\text{nticd}}$, corresponding slices $(\rightarrow_{\text{nticd}})^* (M)$ of sets of nodes M can be computed in linear time. The relation $\rightarrow_{\text{nticd}}$, however, can be of size quadratic in the number of nodes. In Appendix C, I give a generalization of an algorithm based on *DJ-Graphs*[SG95] (i.e.: dominance trees enriched by *join* edges) to the pseudo-forests $<_{\text{SINK}}$, enriched with *conditional* edges (Algorithm 17). The algorithm requires only the postdominance pseudo-forest, but not the corresponding control dependence relation.

NTICD Slices (Figure 8.5) For slices via $\rightarrow_{\text{nticd}}$ (✚), computation time includes the computation of $\rightarrow_{\text{nticd}}$. The computation time for Algorithm 17 (▼) includes the computation of $<_{\text{SINK}}$. I also implemented a variant of Algorithm 17 that is based on $<_{\text{SINK}}$ as represented by postorder numbers, i.e.: as computed by Algorithm 24 on page 406. Marker ■ include computation time of both.

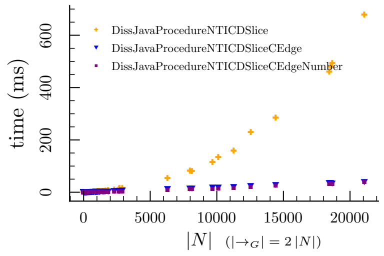
NTSCD Slices (Figure 8.6) For slices via $\rightarrow_{\text{ntscd}}$ (✚), computation time includes the computation of $\rightarrow_{\text{ntscd}}$. Algorithm 17 also works $\rightarrow_{\text{ntscd}}$ slices, by using $<_{\text{MAX}}$ instead of $<_{\text{SINK}}$, and similar in the definition

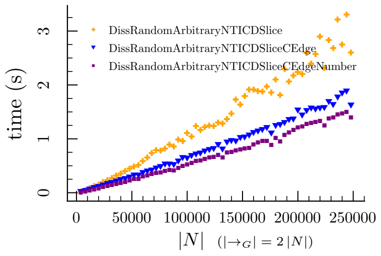
$$n \rightarrow_C m \quad \Leftrightarrow \quad n \rightarrow_G m \;\wedge\; m \notin \text{ipdom}_{\sqsupseteq_{\text{MAX}}}(n)$$

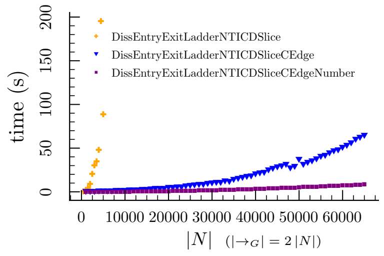
of conditional edges. The computation time (▼) includes the computation of $<_{\text{MAX}}$.

All slices are w.rt. randomly selected noes M, with $|M| = 5$.

Note that in the "bad case" plots, I can show the $|N|$-axis up to $|N| = 65000$, while before in Section 8.3, I could only it up to $|N| = 5000$, since then computation time began to approach and exceed 60*s*.

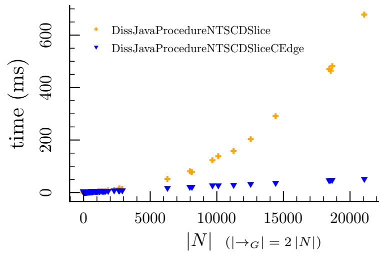


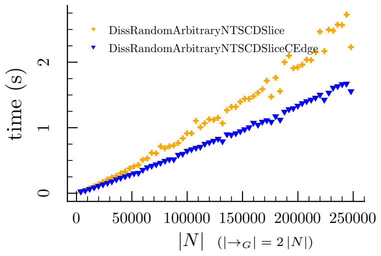


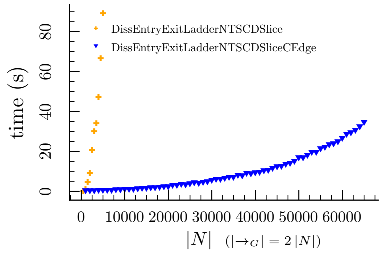


(c) “Bad Case”

Figure 8.5: Computation of $(\rightarrow_{\text{nticd}})^* (M)$







(c) “Bad Case”

Figure 8.6: Computation of $(\rightarrow_{\text{ntscd}})^* (M)$

8.5 Nontermination Insensitive Order Dependence

By the slogan

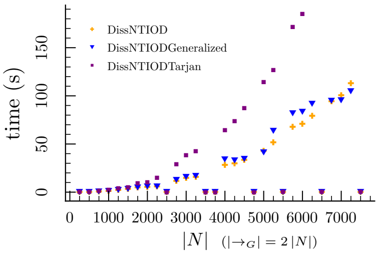
$$\rightarrow_{\text{ntiod}} \quad \approx \quad \left(\sum_{M \in \mathbb{M}} |M| \right) \times \rightarrow_{\text{nticd}}$$

from Subsection 6.7.1, computation of $\rightarrow_{\text{ntiod}}$ for a graph G can be reduced to the computation of $\rightarrow_{\text{nticd}}$ on subgraphs of G. These subgraphs consists of the graphs *control sinks* M (and the those nodes between M and those nodes n of which M are immediate postdominators). Specifically, these subgraphs all have a unique exit node n_x, and hence there it holds that $\rightarrow_{\text{nticd}} = \rightarrow_{\text{cd}}$. This means that classics algorithms[LT79; Cyt+91] can be used. Their computation time form the baseline (■) in Figure 8.7.

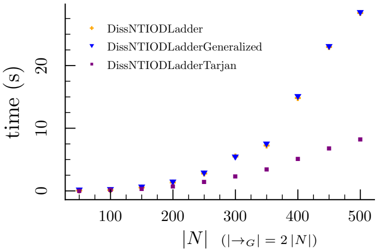
Also in Subsection 6.7.1, I introduced two schemes (Observation 6.7.7 shown as ✚, and Observation 6.7.8 shown as ▼) that avoided full computation of $<_{\text{SINK}}$ for every node $m \in M$ in some control sink M of G. These are based on phase $\text{SINK}_{\text{down}}$ from Algorithm 19.

The plot in Figure 8.7b contains data points (✚, ▼ and ■) near the $|N|$-axis. These indicate that for the given graph (with size $|N|$), the relation $\rightarrow_{\text{ntiod}}$ is either empty or very small (i.e.: there are no non-trivial control sinks in G, or they are small). In other words: aside from *one* computation of $<_{\text{SINK}}$ for the whole graph, my algorithms for $\rightarrow_{\text{ntiod}}$ require additional computation only insofar it contains non-trivial control sinks.

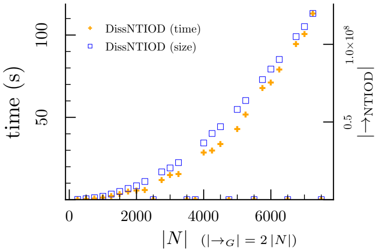
This can also be seen in Figure 8.7d, in which the execution time of the computation due to Observation 6.7.7 (✚) is compared with the size $|\rightarrow_{\text{ntiod}}|$ (□) of the computed relation. I conjecture that in practice, they are asymptotically equal, i.e.: in practice, Observation 6.7.7 is asymptotically optimal under all algorithms that compute an explicit representation of $\rightarrow_{\text{ntiod}}$.



(b) Random Graphs



(c) "Bad Case"



(d) Comparison: execution time vs size $|\rightarrow\text{ntiod}|$.

Figure 8.7: Computation of $\rightarrow$ntiod.

In randomly generated graphs (Figure 8.7b), the two more complicated schemes (✚) and (▼) give an advantage over my baseline scheme (■) (Figure 8.7b). In "bad case" ladder graphs (Figure 8.7c), the situation is reversed.

8.6 Nontermination Insensitive Slices

In Section 7.5, I observed that for any given set M of nodes in an arbitrary graph G, I can reduce three different notions of nontermination insensitive slicing to $\rightarrow_{\text{nticd}}$-slicing in modified graphs G_M, as follows:

$$\begin{aligned}
\left(\rightarrow^{G}_{\text{nticd}} \ \dot{\cup} \rightarrow^{\text{G}}_{\text{ntiod}}\right)^{*}(M) &= \left(\rightarrow^{G_M}_{\text{nticd}}\right)^{*}(M) \text{ for } G_M = G_{M\not\rightarrow} \\
\left(\rightarrow^{\text{G}}_{\text{wod}}\right)^{*}(M) &= \left(\rightarrow^{G_M}_{\text{nticd}}\right)^{*}(M) \text{ for } G_M = G^{\rightarrow^{*}M}_{M\not\rightarrow} \\
\text{WCC}_{G}(M) &= \left(\rightarrow^{G_M}_{\text{nticd}}\right)^{*}(M) \text{ for } G_M = G^{M\rightarrow^{*}M}_{M\not\rightarrow}
\end{aligned}$$

These equations each yield, together with Lemma 7.1.1, an algorithm for computing M-slices *directly* from $<^{G_M}_{\text{SINK}}$, without the need to compute $\rightarrow^{G_M}_{\text{nticd}}$.

With regard to performance, these three reduction behave essentially the same. I show the computation time for weak control slices $\text{WCC}_G(M)$ in Figure 8.8. The best algorithm[2] other than mine is from [LKL18] (▼), with which I compare my algorithm (✚) in randomly generated graphs. The Implementation of the algorithm from [LKL18] is my own, and appears (with respect to constant factors) to be faster than the authors original implementation.

[2] that I am aware of

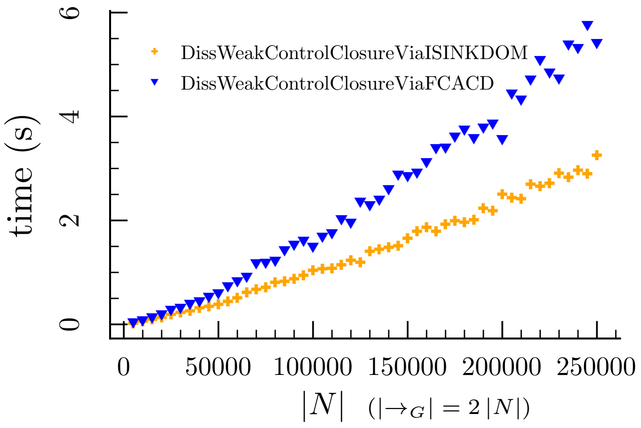


Figure 8.8: Computation of $\mathrm{WCC}_G\,(M)$.

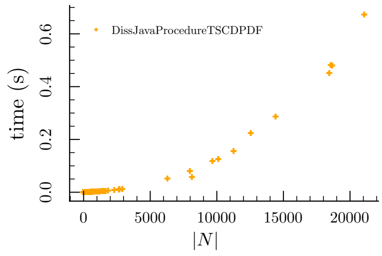
8.7 Timing Sensitive Algorithms

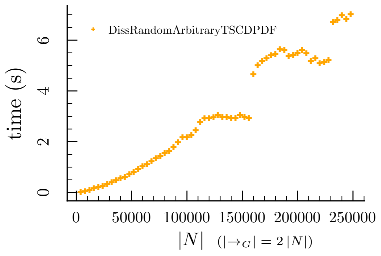
Together, Algorithm 11, 10, 9, 10 and 8 implement timing sensitive control dependence $\rightarrow_{\text{tscd}}$. Timing dependence $\rightarrow_{\text{td}}$ can be computed via Observation 10.2.1 and Algorithm 8.

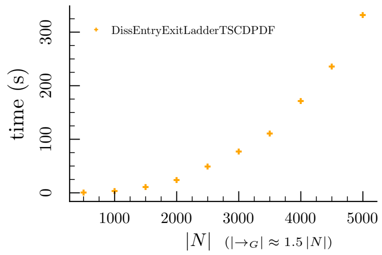
Timing Sensitive Control Dependence The benchmarks include the computation time of all sub-algorithms (✚). Ladder graphs expose non-linear *bad-case* behavior.

Timing Dependence By Observation 10.2.1, for each graph, I compute $|N|$ transitive reductions $<^{G_m}_{\text{TIME}}$ of transitive timing sensitive postdominance $\sqsupseteq^{G_m}_{\text{TIME}}$, for certain subgraphs G_m of the control flow graph G. I use Algorithm 8. The benchmarks include all $|N|$ invocations for each graph (✚).

Timing Sensitive Postdominance Pseudo-Forests I also show benchmarks for the computation of computation of $<_{\text{TIME}}$ only, via Algorithm 8.

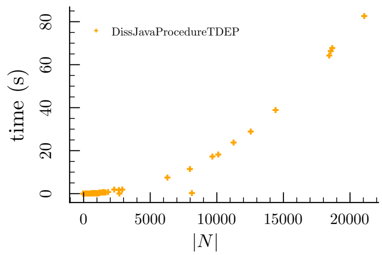


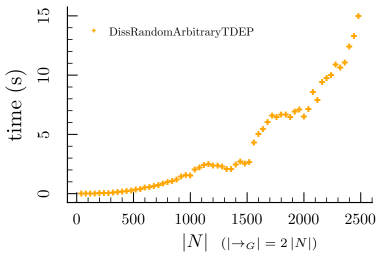


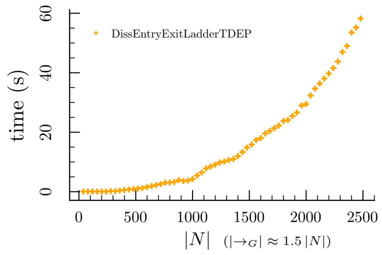


(c) “Bad Case”

Figure 8.9: Computation of $\rightarrow_{\text{tscd}}$.

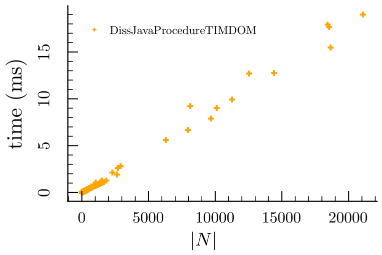


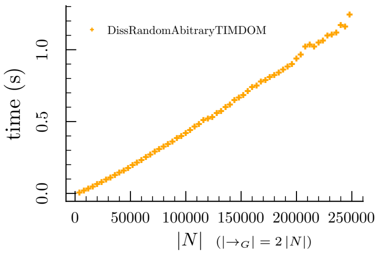


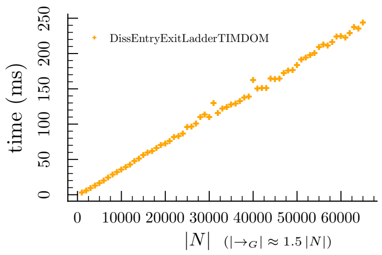


(c) “Bad Case”

Figure 8.10: Computation of $\rightarrow_{\text{timing}}$.







(c) “Bad Case”

Figure 8.11: Computation of $<_{\mathrm{TIME}}$.

Part II

Timing Sensitive Dependency Analysis

Endlich Zeitumstellung zu nehmen (it's time)
Den Wendehals noch schnell umzudrehen
Endlich wieder den Schweinehund ausführen
Zeit umzukehren vor der eigenen Haustür
Höchste Zeit es beim Namen zu nennen
Dass es höchste Zeit ist zum Farbe bekenn', ja

(Dendemann — Zeitumstellung)

9 Timing Sensitive Control Dependence

"Wonderful", the Flatline said, "I never did like to do anything simple when I could do it ass-backwards."

(William Gibson — Neuromancer)

In both Section 6.3 and Section 6.8, I established soundness of slicing by virtue of trace based notions of observation. There, I defined a trace t to be a sequence of *partial edges* $(n, n') \in E \cup (N_x \times \{\bot\})$ that is either finite with

$$t = (n_e, n_1)\,,\ (n_1, n_2)\,,\ \ldots,\ (n_k, n_x)\,,\ (n_x, \bot)$$

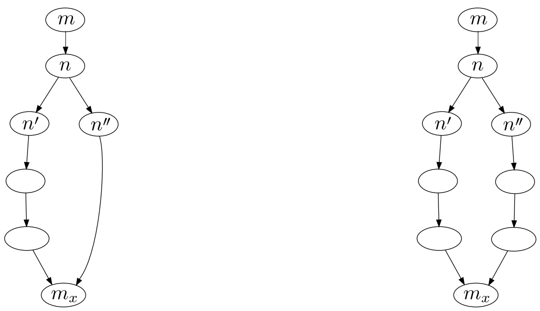
for some exit node n_x, or infinite with

$$t = (n_e, n_1)\,,\ (n_1, n_2)\,,\ \ldots$$

Given a set S of "observable" nodes and a trace t, I defined the S-observation $t\big|_S$ of t to be the sub-sequence of t containing only edges (n, n') with $n \in S$. In terms of an *attacker model*, this means that I assume an attacker to observe exactly those choices made at nodes $n \in S$. Specifically, I assume that an attacker can observe neither the nodes in a subtrace between observable nodes, nor the *time spent* between two observable nodes (i.e: the *length* of the subtrace between two observable nodes). In other words, I make the assumption that the attacker has no clock! As a result, even programs deemed secure under trace equivalence, i.e.: programs such that

$$t_i \sim_S t_{i'}$$

for any two low-equivalent inputs $i \sim_S i'$ may yet have *external timing leaks*. A most trivial example is shown in Figure 9.1a, with observable



(a) A CFG with external timing leak (b) A CFG without timing leak

Figure 9.1: Dependence of execution time of m_x on n.

nodes $S = \{m, m_x\}$. Regardless of the choice made at n, all inputs i, i' starting in m have the same observable trace

$$t_i\big|_S \;=\; (m,n)\,,\;(m_x,\bot) \;=\; t_{i'}\big|_S$$

consisting of two pseudo-edges and terminating in m_x, hence: $t_i \sim_S t_{i'}$. However, if equipped with a suitably precise clock, and assuming a uniform execution time of one time unit u per edge, an attacker will observe m_x after $3u$ have passed if the input i chooses n'' at n, but only after $5u$ if i chooses n' at n, exposing an external timing leak. This becomes obvious if I annotate each edge in traces t_i, t_i' with its execution time, and then compare their S-observation:

$$\begin{aligned} t_i^{\text{⏱}}\big|_S &= (m,n)\text{⏱}[0]\,,\;(m_x,\bot)\text{⏱}[3] \\ &\neq (m,n)\text{⏱}[0]\,,\;(m_x,\bot)\text{⏱}[5] \;=\; t_{i'}^{\text{⏱}}\big|_S \end{aligned}$$

On the other hand, the program in Figure 9.1b has no timing leak. That is, even if I annotate each edge in the observable trace with its execution time, all inputs i, i' starting in m have the same observable *clocked* trace

$$t_i^{\text{⏱}}\big|_S \;=\; (m,n)\text{⏱}[0]\,,\;(m_x,\bot)\text{⏱}[5] \;=\; t_{i'}^{\text{⏱}}\big|_S$$

In this chapter, I fully develop an approach for *timing sensitive* slicing. Unlike nontermination (in)-sensitive slicing, which in the general case required additional ternary notions of dependence, I propose a single (binary) notion of *timing sensitive* control dependence $\rightarrow_{\text{tscd}}$ which will result in slices that are both sound and minimal with regard to observational equivalence of *clocked* traces, justifying the slogan

$$\text{Timing Sensitive IFC} = (\rightarrow_{\text{tscd}} \cup \rightarrow_{\text{data}})^*$$

I will propose an efficient algorithm for the computation of $\rightarrow_{\text{tscd}}$ which will *almost, bot not quite* follow the development of algorithms for $\rightarrow_{\text{nticd}}$ and $\rightarrow_{\text{ntscd}}$ I exercised in Chapter 3 and Chapter 5. Specifically, I will

1. Propose a notion $\sqsupseteq_{\text{TIME[FIRST]}}$ of *timing sensitive* postdominance
2. Give a least fixed point characterization of $\sqsupseteq_{\text{TIME[FIRST]}}$
3. Propose a notion $\rightarrow_{\text{tscd}}$ of *timing sensitive* control dependence. It will be based on $\sqsupseteq_{\text{TIME[FIRST]}}$ *almost* the same way that $\rightarrow_{\text{ntscd}}$ is based on $\sqsupseteq_{\text{MAX}}$.

I will (and in general: can) *not* give an algorithm to compute a transitive, reflexive reduction $>_{\text{TIME[FIRST]}}$ of $\sqsupseteq_{\text{TIME[FIRST]}}$. Instead, I propose another notion $\sqsupseteq_{\text{TIME}}$ of timing sensitive postdominance that is *almost* the same as $\sqsupseteq_{\text{TIME[FIRST]}}$. Then, I will

5. Give a least fixed point characterization of $\sqsupseteq_{\text{TIME}}$
6. Propose an algorithm to compute a transitive, reflexive reduction $>_{\text{TIME}}$ of $\sqsupseteq_{\text{TIME}}$. The Algorithm will be *almost* the same as Algorithm 5 for $>_{\text{MAX}}$.
7. Propose an auxiliary notion of F_n of "fuel available" at nodes n which will allow me to characterize $\sqsupseteq_{\text{TIME[FIRST]}}$ in terms of $\sqsupseteq_{\text{TIME}}$. Informally:

$$\sqsupseteq_{\text{TIME[FIRST]}} \quad = \quad \sqsupseteq_{\text{TIME}} \; + \; \text{F}$$

8. Propose a "post processing" algorithm that — given $>_{\text{TIME}}$ — computes F.

9. Characterize — in terms of $<_{\text{TIME}}$ and F — for very node n the set $\text{ipdom}_{\sqsupseteq_{\text{TIME[FIRST]}}}(n)$ of immediate $\sqsupseteq_{\text{TIME[FIRST]}}$ postdominators of n.

10. Give an algorithm (similar to the generalized Cytron Algorithm from Chapter 3) that — given $\text{ipdom}_{\sqsupseteq_{\text{TIME[FIRST]}}}$ — computes the timing sensitive postdominance frontiers and hence: $\rightarrow_{\text{tscd}}$.

9.1 Timing Sensitive Control Dependence

In Figure 9.1a, the node m_x does nontermination sensitively postdominate n ($m_x \sqsupseteq_{\text{MAX}} n$) because any maximal path starting in either G-successor of n must contain m_x (i.e.: both $m_x \sqsupseteq_{\text{MAX}} n'$ and $m_x \sqsupseteq_{\text{MAX}} n''$), and so must any maximal path starting in n. Remember that $\sqsupseteq_{\text{MAX}}$ was defined via

$$m \sqsupseteq^{G}_{\text{MAX}} n \quad \Leftrightarrow \quad \forall \pi \in {}_{n}\Pi^{G}_{\text{MAX}}.\ m \in \pi$$

where ${}_{n}\Pi^{G}_{\text{MAX}}$ is the set of maximal G-paths starting in n.

In order to account for the different *timing* of the (first) occurrence of m_x in maximal paths starting in n, my following definition is completely natural:

Definition 9.1.1. Let G be any CFG, n, m any nodes in G. Given any path

$$\pi = m_0, m_1, m_2, \ldots$$

I say that m appears in π at position k iff $m = m_k$, and write $m \in^k \pi$. If additionally, $m_i \neq m$ for all $i < k$, I say that m *first* appears in π at position k, and write $m \in^{k}_{\text{FIRST}} \pi$.

Furthermore, I say that m timing-sensitively postdominates n at position $k \in \mathbb{N}$ in G iff on all maximal G-paths starting in n, m first appears at position k. I omit "in G" whenever possible, and just say that m timing-sensitively postdominates n iff this is the case for some k. Formally:

$$\begin{aligned} m \sqsupseteq^{k \text{ in } G}_{\text{TIME[FIRST]}} n \quad &\Leftrightarrow \quad \forall \pi \in {}_{n}\Pi^{G}_{\text{MAX}}.\ m \in^{k}_{\text{FIRST}} \pi \\ m \sqsupseteq^{G}_{\text{TIME[FIRST]}} n \quad &\Leftrightarrow \quad \forall \pi \in {}_{n}\Pi^{G}_{\text{MAX}}.\ m \in^{k}_{\text{FIRST}} \pi \text{ for some } k \in \mathbb{N} \end{aligned}$$

Remark 9.1.1. Obviously, given m and n, the k such that $m \sqsupseteq^{k}_{\text{TIME[FIRST]}} n$ (if it exists!) is unique.

Following Definition 3.1.2 on page 16, but taking into account that $\cdot \sqsupseteq^{\cdot}_{\text{TIME[FIRST]}} \cdot$ is a *ternary* relation, I can immediately define the following timing sensitive notion of control dependence:

Definition 9.1.2. Let G be any CFG, n, m any nodes in G. Then m is said to be *timing sensitively control-dependent* on n, written $n \rightarrow_{\text{tscd}} m$, if there exists G successors n_l and n_r of n, and some $k \in \mathbb{N}$ such that $m \sqsupseteq^{k}_{\text{TIME[FIRST]}}$-post dominates n_l, but *not*: $m \sqsupseteq^{k}_{\text{TIME[FIRST]}}$-post dominates n_r. Formally: $n \rightarrow_{\text{tscd}} m \Leftrightarrow$

$$\begin{array}{rl} & m \sqsupseteq^{k}_{\text{TIME[FIRST]}} n_l \quad \text{and} \\ \neg & m \sqsupseteq^{k}_{\text{TIME[FIRST]}} n_r \end{array}$$

for some $k \in \mathbb{N}$ and n_l, n_r such that $n \rightarrow_G n_l$ and $n \rightarrow_G n_r$.

Remember from Theorem 5.1.2 that $\sqsupseteq_{\text{MAX}}$ is the least fixed point of the following rule system D

$$\frac{}{n \sqsupseteq n}\,\mathsf{D}^{\text{self}} \qquad \frac{\forall p \rightarrow_G x.\, m \sqsupseteq x \qquad p \rightarrow^{*}_{G} m}{m \sqsupseteq p}\,\mathsf{D}^{\text{suc}}$$

in the lattice $\left(2^{N\times N}, \subseteq\right)$.

Similarly, the ternary relation $\sqsupseteq_{\text{TIME[FIRST]}}$ is the least fixed point of the rule system $\mathsf{T}_{\text{FIRST}}$ in the underlying lattice $\left(2^{N\times\mathbb{N}\times N}, \subseteq\right)$.

Proposition 9.1.1. Let G be a CFG and $\mathsf{T}_{\text{FIRST}}$ be the rule-system[1]

$$\frac{}{n \sqsupseteq^{0} n}\,\mathsf{T}^{\text{self}}_{\text{FIRST}} \qquad \frac{\forall p \rightarrow_G x.\, m \sqsupseteq^{k} x \qquad m \neq p \qquad p \rightarrow^{+}_{G} m}{m \sqsupseteq^{k+1} p}\,\mathsf{T}^{\text{suc}}_{\text{FIRST}}$$

Then $\sqsupseteq_{\text{TIME[FIRST]}} = \mu \mathsf{T}_{\text{FIRST}}$.

[1] over a ternary relation $\cdot \sqsupseteq^{\cdot} \cdot$

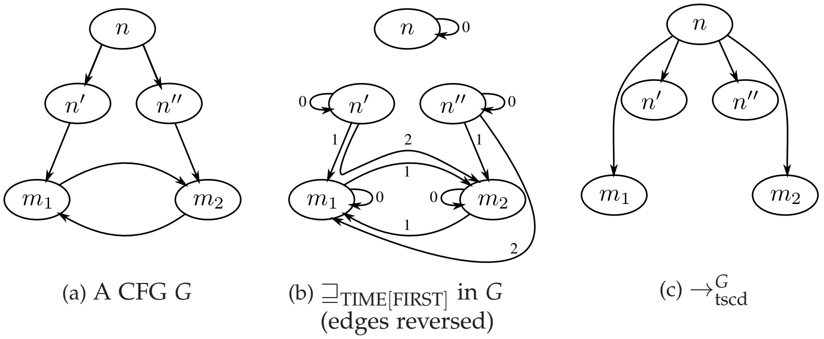
Remark 9.1.2. The condition $p \rightarrow_G^* m$ is redundant for nodes p that have *some* successor x, since I only consider the least (but never: the greatest) fixed point of $\mathsf{T}_{\text{FIRST}}$.

The timing sensitive postdominance for the CFG from the earlier example in Figure 5.1a is shown in Figure 9.3b. Figure 9.3c and Figure 9.3d show the corresponding non-termination sensitive and timing sensitive control dependencies. Note, for example, that $7 \rightarrow_{\text{tscd}} 8$ because a choice $7 \rightarrow_G 11$ can delay node 8, but in contrast: $\neg\, 7 \rightarrow_{\text{ntscd}}^* 8$, because no choice at node 7 can *prevent* node 8 from being executed. It is *not* the case that, in general, $n \rightarrow_{\text{ntscd}} m$ implies $n \rightarrow_{\text{tscd}} m$. For example: $2 \rightarrow_{\text{ntscd}} 8$, but $\neg\, 2 \rightarrow_{\text{tscd}} 8$. What *does* hold here is $2 \rightarrow_{\text{tscd}}^* 8$ via $2 \rightarrow_{\text{tscd}} 7 \rightarrow_{\text{tscd}} 8$. In fact, timing sensitive control independence is *transitively* a stricter requirement than non-termination sensitive control independence:

Observation 9.1.1. Let $G = (N, E)$ be any CFG, and $M \subseteq N$ any set of nodes. Then the timing sensitive backward slice of M contains the nontermination sensitive backward slice of M:

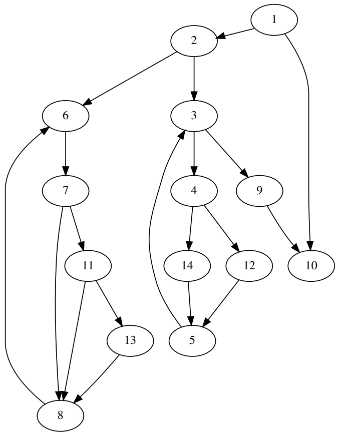
$$\begin{aligned} (\rightarrow_{\text{tscd}})^* (M) &\supseteq (\rightarrow\text{ntscd} \cup \rightarrow\text{ntsod})^* (M) \\ &= (\rightarrow\text{ntscd} \cup \rightarrow\text{dod})^* (M) \end{aligned}$$

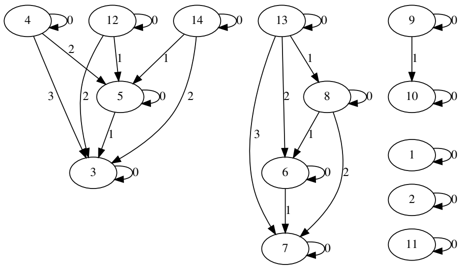
It is worth noting that the $\rightarrow$tscd slice in Observation 9.1.1 does *not* require a timing sensitive analogue of the relation $\rightarrow$ntsod. Recall that the necessity of either $\rightarrow$dod or $\rightarrow$ntsod was motivated by the canonical irreducible graph from Figure 6.1 on page 66. Essentially the same CFG is shown in Figure 9.2. The problem with $\rightarrow$ntscd was that neither m_1 nor m_2 is nontermination sensitively control dependent on n, yet the decision at n determines which node is observed next. In contrast, both m_1 and m_2 *are* timing-sensitively control dependent on n (e.g.: $n \rightarrow_{\text{tscd}} m_1$ because $m_1 \sqsupseteq^1_{\text{TIME[FIRST]}} n'$, but $\neg\, m_1 \sqsupseteq^1_{\text{TIME[FIRST]}} n''$, and also: $m_1 \sqsupseteq^2_{\text{TIME[FIRST]}} n''$, but $\neg\, m_1 \sqsupseteq^2_{\text{TIME[FIRST]}} n'$).

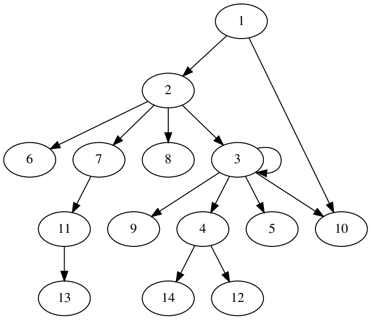


n
n'
n''
m_1
m_2
(a) A CFG G
0
1
2
(b) $\sqsupseteq_{\text{TIME[FIRST]}}$ in G (edges reversed)
(c) $\rightarrow^{G}_{\text{tscd}}$

Figure 9.2: The canonical irreducible graph, where neither $n \rightarrow_{\text{ntscd}} m_1$ nor $n \rightarrow_{\text{ntscd}} m_2$.


(a) A CFG


(b) Its relation $\sqsupseteq_{\text{TIME[FIRST]}}$ (edges reversed)


(c) Its non-termination sensitive control dependence $\rightarrow_{\text{ntscd}}$

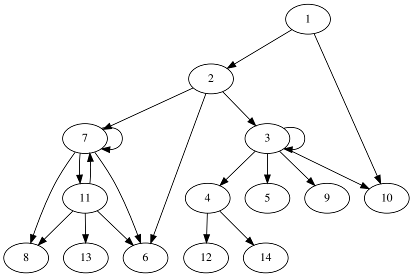
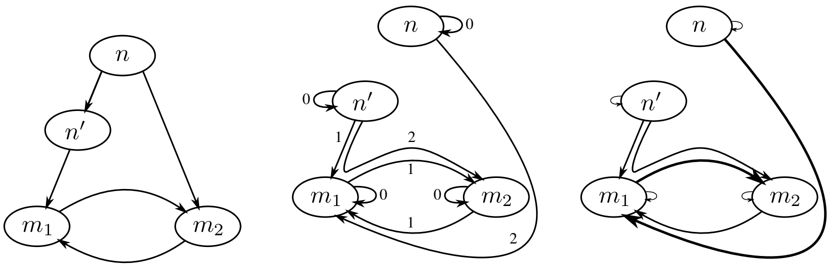

(d) Its timing sensitive control dependence $\rightarrow_{\text{tscd}}$

Figure 9.3: Timing sensitive postdominance. Edges $n \xrightarrow{k} m$ indicate $m \sqsupseteq^{k}_{\text{TIME[FIRST]}} n$.



(a) A CFG G

(b) $\cdot \; \dot{\sqsupseteq}_{\text{TIME[FIRST]}} \; \cdot$ edges reversed

(c) $\cdot \; \sqsupseteq_{\text{TIME[FIRST]}} \; \cdot$ edges reversed

Figure 9.4: An irreducible graph with *intransitive* $\cdot \; \sqsupseteq_{\text{TIME[FIRST]}} \; \cdot$

9.2 Timing Sensitive Post Postdominance Frontiers

In order to develop efficient algorithms for the computation of timing sensitive postdominance $\sqsupseteq_{\text{TIME[FIRST]}}$ and timing sensitive control-dependence $\rightarrow_{\text{tscd}}$, I first recall that algorithms I previously developed for *nontermination* sensitive postdominance and control dependence (i.e.: for $\sqsupseteq_{\text{MAX}}$ and $\rightarrow_{\text{ntscd}}$) heavily relied on the fact that $\sqsupseteq_{\text{MAX}}$ is *transitive*:

1. Transitivity of $\sqsupseteq_{\text{MAX}}$ allowed me to efficiently compute and represent $\sqsupseteq_{\text{MAX}}$ in form of its transitive reduction $>_{\text{MAX}}$. Here, $<_{\text{MAX}}$ turned out to be a pseudo-forest.

2. Transitivity of $\sqsupseteq_{\text{MAX}}$, and the fact that

$$\text{ipdom}^{*}_{\sqsupseteq_{\text{MAX}}} = \sqsupseteq_{\text{MAX}}$$

   allowed me to use Algorithm 1 to efficiently compute $\rightarrow_{\text{ntscd}}$ (see Definition 3.2.7) via the nontermination sensitive postdominance frontier $\text{PDF}_{\sqsupseteq_{\text{MAX}}}$.

Disregarding for now the fact that $\rightarrow_{\text{tscd}}$ is defined in terms of the *ternary* relation $\cdot \ \sqsupseteq^{\cdot}_{\text{TIME[FIRST]}} \ \cdot$, and not in terms of its (binary) "$\exists k.$ -closure" $\cdot \ \sqsupseteq_{\text{TIME[FIRST]}} \ \cdot$, let me investigate first if $\cdot \ \sqsupseteq_{\text{TIME[FIRST]}} \ \cdot$ is — in general — transitive. To do this, consider the (irreducible) CFG in Figure 9.4a. Here, every maximal path starting in n first reaches m_1 after two steps, hence $m_1 \sqsupseteq_{\text{TIME[FIRST]}} n$. Also, every maximal path starting in m_1 first reaches m_2 after one step, hence $m_2 \sqsupseteq_{\text{TIME[FIRST]}} m_1$. But it is for *no* number k of steps the case that $m_2 \sqsupseteq^{k}_{\text{TIME[FIRST]}} n$, hence: $\neg\ m_1 \sqsupseteq_{\text{TIME[FIRST]}} n$. In summary, $\cdot \ \sqsupseteq_{\text{TIME[FIRST]}} \ \cdot$ here is *not* transitive.

On the other hand, situations such as that in Figure 9.4 are the *only* in which $\cdot \ \sqsupseteq_{\text{TIME[FIRST]}} \ \cdot$ is not transitive:

Observation 9.2.1. Let G be any *reducible* CFG, and write $\sqsupseteq$ for $\cdot \sqsupseteq_{\text{TIME[FIRST]}} \cdot$. Then $\sqsupseteq$ is transitive.

Also, if there exists a unique exit node n_x, then $\cdot \ \sqsupseteq_{\text{TIME[FIRST]}} \ \cdot$ is transitive even for irreducible CFG.

Observation 9.2.2. Let G be any CFG with unique exit node n_x, and write $\sqsupseteq$ for $\cdot \sqsupseteq_{\text{TIME[FIRST]}} \cdot$. Then $\sqsupseteq$ is transitive.

Since I know now that $\cdot \ \sqsupseteq_{\text{TIME[FIRST]}} \ \cdot$ is "usually" transitive, I am encouraged to work towards a modification of Algorithm 1 which allows me to compute the timing sensitive postdominance frontier $\text{PDF}_{\sqsupseteq_{\text{TIME[FIRST]}}}$. But first, I need to ensure that $\text{PDF}_{\sqsupseteq_{\text{TIME[FIRST]}}}$ will actually allow me to determine $\rightarrow_{\text{tscd}}$. Remember that for $m \neq n$, I simply had

$$n \in \text{PDF}_{\sqsupseteq_{\text{MAX}}}(m) \ \Leftrightarrow \ n \rightarrow_{\text{ntscd}} m$$

which was obvious from the definition of $\rightarrow_{\text{ntscd}}$, which is in terms of the binary relation $\sqsupseteq_{\text{MAX}}$.

In order to obtain the analogous result for $\rightarrow_{\text{tscd}}$, I first need to "conservatively" redefine the notion $\text{PDF}_{\sqsupseteq}$ of $\sqsupseteq$-postdominance in order to obtain a notion appropriate for non-transitive relations $\sqsupseteq$. Remember that in Definition 3.2.2 on page 21, I defined for any binary relation $\sqsupseteq$:

$$\mathrm{PDF}_{\sqsupseteq}(m) = \left\{ n \;\middle|\; \begin{array}{rr} & \neg\, m \text{ 1-}\sqsupseteq n \\ \text{for some } n' \text{ s.t. } n \rightarrow_G n' : & m \sqsupseteq n' \end{array} \right\}$$

Syntactically, I will stick with this definition, but I will modify the notion of 1-$\sqsupseteq$-postdominance. The new definition is

Definition 9.2.1 (1-$\sqsupseteq$-Postdominance, redefinition)**.** Given a relation $\sqsupseteq \subseteq N \times N$, a node $x \in N$ is said to 1-$\sqsupseteq$-postdominate z if $\boxed{x \sqsupseteq z \text{ and}}$ there exists some node y such that

$$x \sqsupset y \sqsupseteq z$$

The only change is the new requirement $\boxed{x \sqsupseteq z}$, which of course was redundant up to this chapter, since any relation $\sqsupseteq$ I considered (i.e.: $\sqsupseteq_{\mathrm{POST}}$, $\sqsupseteq_{\mathrm{MAX}}$ and $\sqsupseteq_{\mathrm{SINK}}$) was transitive. Implicitly, this change also affects immediate $\sqsupseteq$-postdominance $\mathrm{ipdom}_{\sqsupseteq}$ — see Definition 3.2.1 on 19.

Theorem 9.2.1. Let G be any CFG, and $n \neq m$ two nodes in G. Then

$$n \in \mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}(m) \;\Leftrightarrow\; n \rightarrow_{\mathrm{tscd}} m$$

Proof: In this proof, I write $\sqsupseteq$ for $\sqsupseteq_{\mathrm{TIME[FIRST]}}$, and $x \sqsupseteq^k y$ for $x \sqsupseteq^k_{\mathrm{TIME[FIRST]}} y$, and note that $\sqsupseteq$ is reflexive.

I begin with the forward implication, and assume $n \in \mathrm{PDF}_{\sqsupseteq}(m)$. From $m \neq n$ and since $\neg\, m$ 1-$\sqsupseteq$ n, I conclude that $\neg\, m \sqsupseteq n$. Since $m \neq n$, this just means that for *all* $k \in \mathbb{N}$ there must exist some G-successor n'' such that

$$\neg\, m \sqsupseteq^k n''$$

But at the same time, from $n \in \mathrm{PDF}_{\sqsupseteq}(m)$ I obtain some G-successor n' of n with

$$m \sqsupseteq^{k} n'$$

for *some* k and hence: $n \rightarrow_{\text{tscd}} m$.

For the reverse implication, I can assume

$$\begin{array}{rll} & m \sqsupseteq^{k} & n_l \\ \neg & m \sqsupseteq^{k} & n_r \end{array}$$

for some G-successors n_l, n_r of n and some $k \in \mathbb{N}$. Since I do have $m \sqsupseteq n_l$, all I need to show is $\neg\ m$ 1-$\sqsupseteq$ n. I assume the opposite in order to derive a contradiction. Given the new definition of m 1-$\sqsupseteq$ n, I then have some k' such that $m \sqsupseteq^{k'} n$. Because $m \neq n$, I have $k' > 0$ and both

$$\begin{array}{l} m \sqsupseteq^{k'-1} n_l \quad \text{and} \\ m \sqsupseteq^{k'-1} n_r \end{array}$$

But this means $k = k' - 1$ in contradiction of $\neg m \sqsupseteq^{k} n_r$. □

The postdominance frontier algorithm from [Cyt+91], and my generalization of that algorithm in Chapter 3, was based on the separation of the postdominance frontier $\mathrm{PDF}_{\sqsupseteq}(x)$ into its *local* part $\mathrm{PDF}^{\text{local}}_{\sqsupseteq}(x)$, and the parts $\mathrm{PDF}^{\text{up}}_{\sqsupseteq}(z)$ contributed to $\mathrm{PDF}_{\sqsupseteq}(x)$ by nodes z for which x is an immediate postdominator. In order to apply the same idea for timing sensitive postdominance $\sqsupseteq_{\text{TIME[FIRST]}}$, I can keep the definition for the local part $\mathrm{PDF}^{\text{local}}_{\sqsupseteq}(x)$ (changed only implicitly due to the redefinition of 1-$\sqsupseteq$-postdominance), which is (see: Definition 3.2.3 on page 21):

$$\mathrm{PDF}^{\text{local}}_{\sqsupseteq}(x) = \left\{ y \;\middle|\; \begin{array}{c} \neg\ x \text{ 1-}\sqsupseteq y \\ y \rightarrow_G x \end{array} \right\}$$

I do, however, have to be more discriminate when considering the up-contributions, since due to the implicit redefinition of immediate postdominance, nodes that are $\sqsupseteq_{\text{TIME[FIRST]}}$-"equivalent" (i.e.: nodes

x, x' such that both $x \sqsupseteq_{\text{TIME[FIRST]}} x'$ and $x \sqsupseteq_{\text{TIME[FIRST]}} x'$) no longer necessarily are immediate postdominators of the same nodes z. Remember that the definition for immediate postdominators reads:

$$\text{ipdom}_{\sqsupseteq}(z) = \left\{ x \;\middle|\; \begin{array}{c} x \ 1\text{-}\sqsupseteq z \\ \forall x' \in N.\ x' \ 1\text{-}\sqsupseteq z \implies x' \sqsupseteq x \end{array} \right\}$$

Specifically, I do *not* use the Definition 3.2.4 which read:

$$\text{PDF}^{\text{up}}_{\sqsupseteq}(z) = \left\{ y \in \text{PDF}_{\sqsupseteq}(z) \;\middle|\; \forall x \in \text{ipdom}_{\sqsupseteq}(z)\,.\ \neg\ x \ 1\text{-}\sqsupseteq y \right\}$$

Instead, I use the following:

Definition 9.2.2 ($\sqsupseteq$-Postdominance Frontiers: *up* part for a given immediate postdominator x)**.** Given a CFG $G = (N, E)$, a relation $\sqsupseteq \ \subseteq N \times N$ and nodes $x, z \in N$ such that

$$x \in \text{ipdom}_{\sqsupseteq}(z)$$

the $\sqsupseteq$-postdominance frontiers *up* part $\text{PDF}^{\text{up}}_{\sqsupseteq}(z, x)$ for z given x is defined by

$$\text{PDF}^{\text{up}}_{\sqsupseteq}(z, x) = \left\{ y \in \text{PDF}_{\sqsupseteq}(z) \;\middle|\; \neg\ x \ 1\text{-}\sqsupseteq y \right\}$$

Unfortunately, even this change does not give me the same decomposition of $\text{PDF}_{\sqsupseteq_{\text{TIME[FIRST]}}}(x)$ as I had before (i.e.: as I had in Lemma 3.2.2 on page 22). Specifically, at some nodes x I can only inherit dominance frontier nodes y (from nodes z s.t. x is a immediate $\sqsupseteq_{\text{TIME[FIRST]}}$-postdominator of z) under the additional condition $y \in \sqsubseteq'_x$, for $\sqsubseteq'_x$ as defined as in the following observation:

Observation 9.2.3. Given any CFG $G = (N, E)$ and a node $x \in N$, let

$$\sqsupseteq \ = \ \sqsupseteq_{\text{TIME[FIRST]}}$$

Then:

$$\begin{array}{rcl} \mathrm{PDF}_{\sqsupseteq}(x) & = & \mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(x) \\ \cup & \bigcup\limits_{\{z \mid x \in \mathrm{ipdom}_{\sqsupseteq}(z),\ \neg z \in \mathrm{ipdom}_{\sqsupseteq}(x)\}} & \mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z,x) \\ \cup & \bigcup\limits_{\{z \mid x \in \mathrm{ipdom}_{\sqsupseteq}(z),\ \ z \in \mathrm{ipdom}_{\sqsupseteq}(x)\}} & \mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z,x) \cap \sqsubseteq'_x \end{array}$$

where $\sqsubseteq'_x = \{ y \mid \text{for some } y' \text{ s.t. } y \rightarrow_G y' : \quad x \sqsupseteq y' \}$.

In contrast to the corresponding result (Lemma 3.2.2 on page 22) for transitive postdominance relations, I do not offer a formal proof of Observation 9.2.3. I also do not attempt to give general conditions under which Observation 9.2.3 may hold for (intransitive) postdominance relations other than $\sqsupseteq_{\mathrm{TIME[FIRST]}}$. I also do not offer any such conditions for the two following simplifications of $\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(x)$ and $\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z,x)$. I do want to note, however, that while $\sqsupseteq_{\mathrm{TIME[FIRST]}}$ is in general not transitive, it *does* fulfill the other two earlier conditions of being *closed under* $\rightarrow_G$ and *lacking joins*.

Observation 9.2.4. Given any CFG G, let

$$\sqsupseteq \; = \; \sqsupseteq_{\mathrm{TIME[FIRST]}}$$

Then $\sqsupseteq$ is closed under $\rightarrow_G$, and

$$\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(x) = \left\{ y \;\middle|\; \begin{array}{r} \neg\, x \in \mathrm{ipdom}_{\sqsupseteq}(y) \\ y \rightarrow_G x \end{array} \right\}$$

Observation 9.2.5. Given any CFG G, let

$$\sqsupseteq \; = \; \sqsupseteq_{\mathrm{TIME[FIRST]}}$$

$$\frac{\neg\ x \in \mathrm{ipdom}_{\sqsupseteq}(y) \qquad y \to_G x}{y \in \mathrm{PDF}_{\sqsupseteq}(x)}\ \mathrm{PDF}^{\mathrm{local}}$$

$$\frac{\begin{array}{ccc} & \neg\ x \in \mathrm{ipdom}_{\sqsupseteq}(y) & \\ x \in \mathrm{ipdom}_{\sqsupseteq}(z) & \neg\ z \in \mathrm{ipdom}_{\sqsupseteq}(x) & y \in \mathrm{PDF}_{\sqsupseteq}(z) \end{array}}{y \in \mathrm{PDF}_{\sqsupseteq}(x)}\ \mathrm{PDF}^{\mathrm{up}}_{1}$$

$$\frac{\begin{array}{ccc} y \to_G y' & \neg\ x \in \mathrm{ipdom}_{\sqsupseteq}(y) & x \sqsupseteq y' \\ x \in \mathrm{ipdom}_{\sqsupseteq}(z) & z \in \mathrm{ipdom}_{\sqsupseteq}(x) & y \in \mathrm{PDF}_{\sqsupseteq}(z) \end{array}}{y \in \mathrm{PDF}_{\sqsupseteq}(x)}\ \mathrm{PDF}^{\mathrm{up}}_{2}$$

Figure 9.5: A rule system for $\mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$, writing $\sqsupseteq$ for $\sqsupseteq_{\mathrm{TIME[FIRST]}}$.

Then $\sqsupseteq$ lacks joins and is closed under $\to_G$, and given any $x \in \mathrm{ipdom}_{\sqsupseteq}(z)$:

$$\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z, x) = \left\{ y \in \mathrm{PDF}_{\sqsupseteq}(z) \;\middle|\; \neg\ x \in \mathrm{ipdom}_{\sqsupseteq}(y) \right\}$$

Rephrasing observations 9.2.3, 9.2.4 and 9.2.5, I obtain a characterization of $\mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$ as the least fixed point of the monotone functional defined by the rules in Figure 9.5.

In order to obtain an efficient algorithm for $\mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$, I thus need to develop

1. An efficient algorithm for the computation of (an efficient representation of) $\mathrm{ipdom}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$
2. A way to replace the explicit *generate-and-test*

$$y \to_G y',\ x \sqsupseteq_{\mathrm{TIME[FIRST]}} y'$$

   of some y' such in the rule $\mathrm{PDF}^{\mathrm{up}}_{2}$ by an efficient (implicit) check.

9.3 Transitive Timing Sensitive Postdominance

When I developed algorithms for $\mathrm{ipdom}_{\sqsupseteq_{\mathrm{SINK}}}$ and $\mathrm{ipdom}_{\sqsupseteq_{\mathrm{MAX}}}$, I made use of the fact that there, I had

$$\mathrm{ipdom}^{*}_{\sqsupseteq} = \sqsupseteq$$

and $\mathrm{ipdom}_{\sqsupseteq}$ was easily derived from any transitive reduction $>$ of $\sqsupseteq$ (e.g.: Observation 5.2.2 on 44).

With $\sqsupseteq_{\mathrm{TIME[FIRST]}}$, things are different: for example, in the CFG from Figure 9.4a, repeated in Figure 9.6a, I have

$$\mathrm{ipdom}^{*}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}} \neq \sqsupseteq_{\mathrm{TIME[FIRST]}}$$

as is evident from $\sqsupseteq_{\mathrm{TIME[FIRST]}}$ and $\mathrm{ipdom}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$ shown in the same figure. Note, for example, that

$$\begin{aligned} m_2 &\in \mathrm{ipdom}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}(m_1) \text{ and} \\ m_1 &\in \mathrm{ipdom}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}(m_2)\text{, but} \\ &\neg\, m_2 \sqsupseteq_{\mathrm{TIME[FIRST]}} n \end{aligned}$$

As mentioned before, this very same CFG demonstrates that $\cdot \sqsupseteq_{\mathrm{TIME[FIRST]}} \cdot$ is — in general — not transitive, and hence cannot be efficiently represented by some transitive reduction $>$. Similarly, the ternary relation $\cdot \sqsupseteq^{\cdot}_{\mathrm{TIME[FIRST]}} \cdot$ is — in general — not transitive. Here, by transitive I mean the following:

Definition 9.3.1. A ternary relation $\cdot \sqsupseteq^{\cdot} \cdot \subseteq N \times \mathbb{N} \times N$ is *transitive* if

$$\begin{array}{llcll} \text{whenever} & m \sqsupseteq^{k} m' & \text{and} & m' \sqsupseteq^{k'} n \\ \text{then i also have} & m & \sqsupseteq^{k+k'} & n \end{array}$$

In contrast to $\cdot \sqsupseteq^{\cdot}_{\mathrm{TIME[FIRST]}} \cdot$, the following ternary relation *is* transitive:

Definition 9.3.2. I say that m timing-sensitively and *transitively* post-dominates n at position k in G iff on all maximal G-paths starting in n, m appears at position k. I omit "in G" whenever possible. Formally:

$$m \sqsupseteq^{k \text{ in } G}_{\text{TIME}} n \quad \Leftrightarrow \quad \forall \pi \in {}_{n}\Pi^{G}_{\text{MAX}}.\ m \in^{k} \pi$$

Note that the only difference between this definition of $\sqsupseteq_{\text{TIME}}$ and the Definition 9.1.1 of $\sqsupseteq_{\text{TIME[FIRST]}}$ is the use of $\in^{k}$ instead of $\in^{k}_{\text{FIRST}}$, i.e.: $\sqsupseteq_{\text{TIME}}$ only requires m to appear at position k in π, while $\sqsupseteq_{\text{TIME[FIRST]}}$ additionally requires k to be the *first* position in π at which m appears.

Observation 9.3.1. Given any CFG G, the ternary relation $\cdot \sqsupseteq^{\cdot}_{\text{TIME}} \cdot$ is transitive.

Just as for $\sqsupseteq_{\text{TIME[FIRST]}}$, I can give a least fixed point characterization of $\sqsupseteq_{\text{TIME}}$ in the underlying lattice $\left(2^{N \times \mathbb{N} \times N}, \subseteq\right)$.

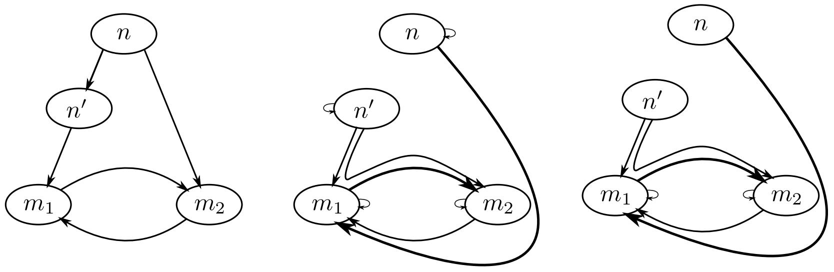
Proposition 9.3.1. Let G be a CFG and T be the rule-system[2]

$$\frac{}{n \sqsupseteq^{0} n}\mathsf{T}^{\text{self}} \qquad \frac{\forall p \to_G x.\ m \sqsupseteq^{k} x \qquad p \to^{+}_{G} m}{m \sqsupseteq^{k+1} p}\mathsf{T}^{\text{suc}}$$

Then $\sqsupseteq_{\text{TIME}} = \mu\mathsf{T}$.

The only difference to $\mathsf{T}_{\text{FIRST}}$ is the omission of the requirement $m \neq p$ in Proposition 9.3.1.

[2] over a ternary relation $\cdot \sqsupseteq^{\cdot} \cdot$



(a) A CFG G (b) $\sqsupseteq_{\text{TIME[FIRST]}}$ (edges reversed) (c) $\text{ipdom}_{\sqsupseteq_{\text{TIME[FIRST]}}}$

Figure 9.6: An irreducible graph with *intransitive* $\cdot \sqsupseteq_{\text{TIME[FIRST]}} \cdot$

Unlike $\sqsupseteq_{\text{TIME[FIRST]}}$, the relation $\sqsupseteq_{\text{TIME}}$ is not finite, so I cannot naively implement $\sqsupseteq_{\text{TIME}}$ to obtain a "reference" implementation. For example, in the CFG from Figure 9.6a, I have

$$\begin{array}{ll}
m_1 \sqsupseteq^{2}_{\text{TIME}} n & \\
 & m_2 \sqsupseteq^{3}_{\text{TIME}} n \\
m_1 \sqsupseteq^{4}_{\text{TIME}} n & \\
 & m_2 \sqsupseteq^{5}_{\text{TIME}} n \\
\quad\dots & \\
 & \quad\dots
\end{array}$$

I can, however, use T to obtain (for any $n \in \mathbb{N}$) a reference implementation for $k \leq n$, i.e. for:

$$\sqsupseteq_{\text{TIME}} \quad \cap \quad N \times \mathbb{N}^{\leq n} \times N$$

using an explicit representation.

The fact that $\sqsupseteq_{\text{TIME}}$ is transitive suggests that I can efficiently represent the full relation $\sqsupseteq_{\text{TIME}}$ using a *transitive reduction* $>$ of $\sqsupseteq_{\text{TIME}}$:

Definition 9.3.3. A ternary relation

$$\cdot > ^{\cdot} \cdot \quad \subseteq \; N \times \mathbb{N} \times N$$

is called a *transitive reduction* of a ternary relation

$$\cdot \sqsupseteq^{\cdot} \cdot \quad \subseteq \; N \times \mathbb{N} \times N$$

iff $\sqsupseteq$ is equal to the reflexive, transitive closure of $>$, and $>$ is minimal w.r.t. this property, i.e.:

1. For any $n, m \in N$ and any $k \in \mathbb{N}$,

$$m \sqsupseteq^k n \quad \Leftrightarrow \quad \begin{array}{l} m >^{k_1} m' >^{k_2} m'' \ldots >^{k_c} n \qquad (c \geq 0) \\ \text{for some nodes } m', m'', \ldots \text{ such that } k = \sum k_i \end{array}$$

   where the case $c = 0$ is understood to mean $k = 0$ and $m = n$, and

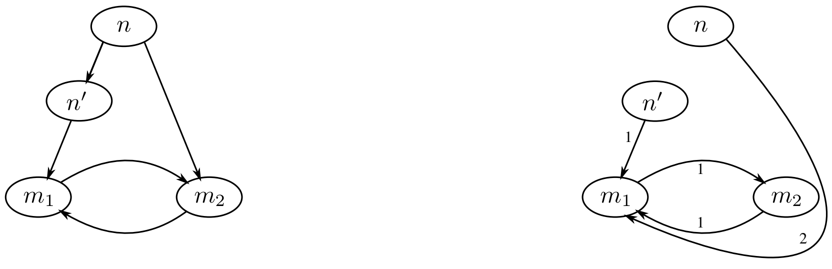
2. $>$ has *minimal size* $|>| \;\leq\; |>'|$ among all such relations $>'$.

In the following, given a ternary relation $>$ (or similar), I will write[3] $n <^k m$ for $m >^k n$. The ternary relation $<_{\text{TIME}}$ corresponding to a ternary transitive reduction $>_{\text{TIME}}$ of $\sqsupseteq_{\text{TIME}}$ for the example CFG is shown in Figure 9.7.

In order to obtain an efficient algorithm for the computation of some transitive reduction $>$ of $\sqsupseteq_{\text{TIME}}$, the following property is crucial:

Proposition 9.3.2. Let G be any CFG, and $>$ a transitive reduction of $\cdot \sqsupseteq^{\cdot}_{\text{TIME}} \cdot$ (and $<$ its inverse, as just defined). Then $<$ is a ($\mathbb{N}$-) labeled pseudo-forest, i.e. for any node n, whenever both $n <^k m$ and $n <^{k'} m'$, I have $k = k'$ and $m = m'$.

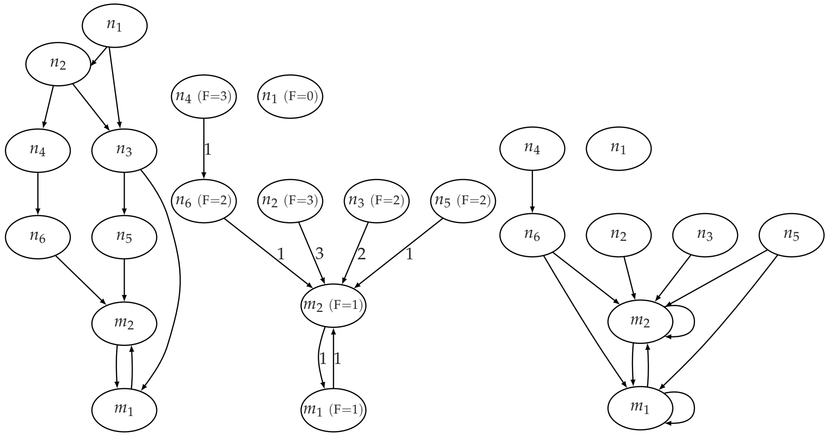
[3] in contrast, I will *never* write $<^k$ or $>^k$ for the kth iteration of some *binary* relation $<$ or $>$



(a) A CFG G

(b) $<_{\text{TIME}}$

Figure 9.7: An irreducible graph with *intransitive* · $\sqsupseteq_{\text{TIME[FIRST]}}$ ·



(a) A CFG G

(b) $<_{\text{TIME}}$

(c) $\text{ipdom}_{\sqsupseteq_{\text{TIME}}}$

Figure 9.8: An irreducible graph with *intransitive* · $\sqsupseteq_{\text{TIME[FIRST]}}$ ·

In order for $\sqsupseteq_{\text{TIME}}$ to be of any use in obtaining an algorithm for $\rightarrow_{\text{tscd}}$, I must now (by Observation 9.2.3) explain how to derive $\text{ipdom}_{\sqsupseteq_{\text{TIME[FIRST]}}}$ from $\sqsupseteq_{\text{TIME}}$.

In Figure 9.8a, I show a somewhat more interesting CFG. First note that $m_1 \sqsupseteq^3_{\text{TIME}} n_3$, but *not:* $m_1 \sqsupseteq^3_{\text{TIME[FIRST]}} n_3$ (nor: $m_1 \sqsupseteq^k_{\text{TIME[FIRST]}} n_3$ for any other k), since I can reach m_1 from n_3 in just $1 \neq 3$ G-

step $n_3 \rightarrow_G m_1$. In terms of "following edges" in $<_{\text{TIME}}$, I can say: the $<_{\text{TIME}}$-path $n_3 <^2_{\text{TIME}} m_2 <^1_{\text{TIME}} m_1$ is valid with respect to $\sqsubseteq_{\text{TIME[FIRST]}}$ only up to m_2.

On the other hand, consider the G-successors n_3 and n_4 of n_2. Both the $<_{\text{TIME}}$-paths $n_3 <^2_{\text{TIME}} m_2$ and $n_4 <^1_{\text{TIME}} m_6 <^1_{\text{TIME}} m_2$ are valid up to m_2, and have the same total "cost" 2, from which I can conclude $m_2 \sqsupseteq^{3=1+2}_{\text{TIME[FIRST]}} n_2$ (adding to the common cost 2 the cost 1 that is due to going from n_2 to any of its G-successor n_3 or n_4).

In general, I make the following observation on the relation of $\sqsupseteq_{\text{TIME[FIRST]}}$ and $\sqsupseteq_{\text{TIME}}$:

Observation 9.3.2. Let G be any CFG. Then for every node n there exists a number $\text{F}_n \in \mathbb{N}$ (the "amount of fuel available at n") such that for all nodes m and any "distance" $k \in \mathbb{N}$ such that $m \sqsupseteq^k_{\text{TIME}} n$, I have

$$m \sqsupseteq^k_{\text{TIME[FIRST]}} n \quad \Leftrightarrow \quad k \leq \text{F}_n$$

In the following, I just write F_n for the least such number.

Let me try to justify the colloquial phrase "amount of fuel available at n" by rephrasing Observation 9.3.2 in terms of a transitive reduction $>_{\text{TIME}}$ of $\sqsupseteq_{\text{TIME}}$: When "starting" at node n with a tank filled with F_n units of fuel, and if following edges $<^f_{\text{TIME}}$ consumes f units of fuel, the nodes m such that $m \sqsupseteq^k_{\text{TIME[FIRST]}} n$ for some k are exactly those nodes I can reach without running out of fuel. Given such m, the number k is exactly the amount of fuel consumed before reaching m.

Rephrasing my remarks on Figure 9.8b, I first note that $m_1 \sqsupseteq^{k=3}_{\text{TIME}} n_3$ (and that $k = 3$ is the *least* such k), but *not:* $m_1 \sqsupseteq^{k=3}_{\text{TIME[FIRST]}} n_3$ (nor: $m_1 \sqsupseteq^{k=k}_{\text{TIME[FIRST]}} n_3$ for any other k), since $k = 3 > 2 = \text{F}_{n_3}$. In other words: attempting to reach m_1 from n_3 in $<_{\text{TIME}}$ fails by running out of fuel, since after starting with $\text{F}_{n_3} = 2$ and traversing the edge $n_3 <^2_{\text{TIME}} m_2$ my tank is empty, and I cannot traverse the edge $m_2 <^1_{\text{TIME}} m_1$ to reach m_1.

On the other hand, I have $m_2 \sqsupseteq^3_{\text{TIME}} n_2$ and $3 \leq 3 = \mathrm{F}_{n_2}$, from which I can conclude $m_2 \sqsupseteq^3_{\text{TIME[FIRST]}} n_2$.

The following observation states that the function $n \mapsto \mathrm{F}_n$ on the nodes of some CFG is easily recoverable from its restriction to nodes "entering" some $<_{\text{TIME}}$-cycle M, i.e.: nodes $n \in N_M$ where (similar to the N_M in Subsection 6.1.1):

$$N_M = \{\, n \in N \mid n \notin M,\ \exists m \in M,\ k \in \mathbb{N}.\ n <^k_{\text{TIME}} m \,\}$$

In Figure 9.8b, the only cycle is $M = \{m_1, m_2\}$, and $N_M = \{n_2, n_3, n_5, n_6\}$.

Observation 9.3.3. Let G be any CFG, $>_{\text{TIME}}$ a transitive reduction of $\sqsupseteq_{\text{TIME}}$, and $n \neq m$ nodes such that $n <^k_{\text{TIME}} m$. Then either

a) n is a node "entering" some $<_{\text{TIME}}$-cycle M (i.e.: $n \in N_M$) and $m \in M$, or

b) neither n nor m are in any $<_{\text{TIME}}$-cycle, and $\mathrm{F}_n = \mathrm{F}_m + k$, or

c) both n and m are in some $<_{\text{TIME}}$-cycle M, and

$$\begin{array}{rl} & m \sqsupseteq^k_{\text{TIME[FIRST]}} n \\ \text{as well as} & n \sqsupseteq^{\mathrm{F}_m}_{\text{TIME[FIRST]}} m \end{array}$$

Remark 9.3.1. In item c) of Observation 9.3.3, the sum $k + \mathrm{F}_m$ is exactly the *circumference* c_M of the $<_{\text{TIME}}$-cycle M, i.e. the sum over all the distance in M:

$$c_M = \sum_{m \in M,\ m <^k_{\text{TIME}} m'} k$$

Note also that for any $m_1, m_2 \in M$ such that $m_1 \neq m_2$, I *always* have

$$\begin{aligned} & m_1 \sqsupseteq^{k}_{\text{TIME[FIRST]}} m_2 \\ \text{as well as} \quad & m_2 \sqsupseteq^{k'}_{\text{TIME[FIRST]}} m_1 \end{aligned}$$

for some k, k' with $k + k' = c_M$.

Let me try to rephrase Observation 9.3.3 in colloquial terms. Assume I want to *start a trip* in $<_{\text{TIME}}$, starting at node n_0. The total fuel available for my trip then is F_{n_0}, and traversing some edge $n <^{k}_{\text{TIME}} m$ will cost k units of fuel. By Observation 9.3.2, whenever I reach some node m after spending k units of fuel, I learn $m \sqsupseteq^{k}_{\text{TIME[FIRST]}} n_0$. Now, item b) and c) in Observation 9.3.3 tell me that

b) if I start my trip at some node n_0 that is not in any $<_{\text{TIME}}$-cycle M, then I will never run out of fuel, *unless* I reach some node $n \in N_M$ from which I will "enter" some $<_{\text{TIME}}$-cycle M. Then at that node n, I am guaranteed to have exactly F_n units of fuel remaining (which may allow me to visit some, but not necessarily all nodes $m \in M$).

c) if I start my trip at in some $<_{\text{TIME}}$-cycle M (i.e.: $n_0 \in M$), then I will run out of fuel *just short of coming back to* n_0, i.e. at the node $m \in M$ such that $m <_{\text{TIME}} n_0$. In other words, I will visit exactly the nodes in M, and every node exactly once.

Being lazy as I am, this means that after fueling up my empty tank at n_0, I do not even need to worry about my fuel consumption, unless I visit some border city $n \in N_M$. At that point, some friendly road sign[4] will assure me: "You have F_n units of fuel remaining". Additionally, another road sign will tell me: "You have enough fuel to visit: $\{m_1, m_2, \ldots\} \subseteq M$". Only *this* road sign will have been put up by $\text{ipdom}_{\sqsupseteq_{\text{TIME[FIRST]}}}$:

[4] valid no matter where exactly I started my journey!

Observation 9.3.4. Given any CFG G, let

$$\sqsupseteq \; = \; \sqsupseteq_{\text{TIME[FIRST]}}$$

Then for any node n

- if $n \in N_M$ for some $<_{\text{TIME}}$-cycle M, then

  $$\text{ipdom}_{\sqsupseteq}(n) = \{\, m_c \mid n <^{k_1}_{\text{TIME}} m_1 <^{k_2}_{\text{TIME}} \dots <^{k_c}_{\text{TIME}} m_c,\ \sum_i k_i \leq \text{F}_n \,\}$$

  where the c are understood to be ≥ 1.
- if $n \in M$ for some $<_{\text{TIME}}$-cycle M, then

  $$\text{ipdom}_{\sqsupseteq}(n) = \begin{cases} \emptyset & \text{if } M = \{n\} \\ M & \text{otherwise} \end{cases}$$

- otherwise
  $$\text{ipdom}_{\sqsupseteq}(n) = \{\, m \mid n <^{k}_{\text{TIME}} m \,\}$$
  which is, by virtue of $<_{\text{TIME}}$ being a (labeled) pseudo-forest, at most a singleton.

In colloquial terms: outside of $<_{\text{TIME}}$-cycles M and border cities N_M, the sign put up by $\text{ipdom}_{\sqsupseteq_{\text{TIME[FIRST]}}}$ will just tell me which node m I will visit next (and not to worry about fuel).

9.4 Algorithms for Timing Sensitive Control Dependence

I ended the previous section by giving a (easy-to-implement) scheme to derive $\mathrm{ipdom}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$ from $<_{\mathrm{TIME}}$ and the *fuel*-mapping $\mathrm{F} = n \mapsto \mathrm{F}_n$. Now, I need to explain how to

1. efficiently compute $<_{\mathrm{TIME}}$
2. efficiently compute F (from $<_{\mathrm{TIME}}$)
3. use the previous result from Observation 9.2.3 to obtain an efficient algorithm for $\mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$ and hence (by Theorem 9.2.1): $\rightarrow_{\mathrm{tscd}}$.

9.4.1 An Algorithm for $<_{\mathrm{TIME}}$

This first task is — in principle — the easiest. I merely need to modify the *least common ancestor* Algorithm 3 from page 47 to support *counting* of the cost of $<_{\mathrm{TIME}}$-paths, and then modify the $\sqsupseteq_{\mathrm{MAX}}$-Algorithm 5 to use the modified least common ancestor algorithm. The result of the latter is shown in Algorithm 8. The most significant modifications are highlighted. I need to omit the check $z \neq x$ because I have to differentiate between an exit node x (i.e.: a node x without G-successor), and a "one node sink" x (i.e.: a node x with only G-successor x). For the former I only have $x \sqsupseteq^{0}_{\mathrm{TIME}} x$, while for the latter, I also have $x \sqsupseteq^{1}_{\mathrm{TIME}} x$, $x \sqsupseteq^{2}_{\mathrm{TIME}} x, \ldots$. Instead of finding the least common ancestors in some (binary) pseudo-forest, I need to find the least common ancestor in a $\mathbb{N}$-labeled (ternary) pseudo-forest, by a call to $\mathsf{lca}_{<}$. Here, $\mathsf{lca}_{<}$ on a set (of successors y of x, and their distance 1 from x) is implemented by iterating over Algorithm 9 (similar to Algorithm 4 on page 48).

The *least common ancestor* for $\mathbb{N}$-labeled pseudo forests like $<_{\mathrm{TIME}}$ is defined with respect to the amount k^n, k^m of "amount of fuel already spent" before reaching nodes n, m.

Definition 9.4.1. Let $<$ be a $\mathbb{N}$-labeled pseudo-forest, n, m two nodes in $<$, and $k^n, k^m \in \mathbb{N}$ two natural numbers. If (z, k) is the least (by comparison of k) pair such that both

$$\begin{array}{rlll} & n <^{k_1} \ldots <^{k_c} z & \text{with} & k = k^n + \sum_i k_i \\ \text{and} & m <^{k'_1} \ldots <^{k'_{c'}} z & \text{with} & k = k^m + \sum_i k'_i \end{array}$$

then (z, k) is the *least common ancestor* of n, m in $<$ with respect to initial “fuel consumption” k^n, k^m, and I write

$$\text{lca}_< \left((n, k^n), (m, k^m) \right) = (z, k)$$

I do not attempt to rigorously defend the Algorithm 9 for the computation of $\text{lca}_< \left((n, k^n), (m, k^m) \right)$. I merely note that the check

$$\mathsf{n}' \in \pi_\mathsf{n} \wedge \mathsf{n}' \notin \pi_\mathsf{m} \wedge |\mathsf{KS}_\mathsf{m}[\mathsf{m}]| > 1$$

is needed to guarantee termination whenever π_n has reached (at n') a different $<$-cycle than π_m is in. The check

$$\max \mathsf{KS}_\mathsf{n}[\mathsf{n}] > \min \mathsf{KS}_\mathsf{m}[\mathsf{n}] \quad \wedge \quad \min \mathsf{KS}_\mathsf{n}[\mathsf{n}] < \max \mathsf{KS}_\mathsf{m}[\mathsf{n}]$$

terminates the search whenever both π_n and π_m have reached the same $<$-cycle M, but are “out of phase” with regard to amount of “fuel consumed” before reaching M, and the circumference c_M of M. The use of a map KS that contains multiple numbers k for a given node n once a $<$-cycle M is reached is certainly not most efficient. I could instead make use of the fact that for such nodes n, I have some k_n such that eventually $k_n + i \cdot c_M \in \mathsf{KS}_\mathsf{n}(n)$ for *all* i.

Also note that Algorithm 8 could (by virtue of the two assertions) be optimized by never re-inserting a node n into the workset once $\mathsf{ITIMEDOM}[n] \neq \bot$. Presumably, a workset-free algorithm (similar to Algorithm 19 for $<_{\text{MAX}}$ on page 397) is possible and may be preferable.

```
Input   : A CFG $G$
Data: A $\mathbb{N}$ labeled pseudo-forest $<$, represented as a map
      $\mathsf{IDOM} : N \hookrightarrow N \times \mathbb{N}$ s.t. $\mathsf{IDOM}\,[n] = (m,k)$ iff $n <^k m$
Output: A transitive reduction $>_{\mathrm{TIME}}$ of $\sqsupseteq_{\mathrm{TIME}}$
begin
    for $x \in N,\ \{z \mid x \to_G z\} = \{z\}$ ~~$, z \neq x$~~ do
        $\mathsf{IDOM}\,[x] \leftarrow (z,1)$
    end
    TIME$_{\mathrm{up}}$
    return $\mathsf{IDOM}$
end
Procedure TIME$_{\mathrm{up}}$
    $\mathsf{workset} \leftarrow \mathrm{COND}_G$
    while $\mathsf{workset} \neq \emptyset$ do
        $x \leftarrow \mathit{remove}(\mathsf{workset})$
        $(z,k) \leftarrow \mathsf{lca}_{<}\ (\{\ (y,1)\ \mid x \to_G y\ \})$
        assert $(z,k) \neq \mathsf{IDOM}[x] \ \Rightarrow\ (z,k) \neq \bot$
        assert $(z,k) \neq \mathsf{IDOM}[x] \ \Rightarrow\ \mathsf{IDOM}[x] = \bot$
        if $(z,k) \neq \mathsf{IDOM}[x]$ then
            $\mathsf{workset} \leftarrow \mathsf{workset} \cup \{n \in \mathrm{COND} \mid n \neq x, \exists n \to_G y.\ y <^* x\}$
            $\mathsf{IDOM}\,[x] \leftarrow (z,k)$
        end
    end
end
```

Algorithm 8: An algorithm for the computation of $<_{\mathrm{TIME}}$. Here, $y <^* x$ means: $x = y \ \vee\ y <^{k_1} \ldots <^{k_c} x$

```
Input: A $\mathbb{N}$ labeled pseudo-forest $<$, represented as a map
       $\mathsf{IDOM} : N \hookrightarrow N \times \mathbb{N}$ s.t. $\mathsf{IDOM}[n] = (m, k)$ iff $n <^k m$
Input: Numbers $\mathsf{k}^{\mathsf{n}}_0$, $\mathsf{k}^{\mathsf{m}}_0 \in \mathbb{N}$ and nodes $\mathsf{n}_0$, $\mathsf{m}_0$
Output: $\mathsf{lca}_<((\mathsf{n}_0, \mathsf{k}^{\mathsf{n}}_0), (\mathsf{m}_0, \mathsf{k}^{\mathsf{m}}_0))$ if it exists, or $\perp$ otherwise.
return $\mathsf{lca}((\mathsf{n}_0, \mathsf{k}^{\mathsf{n}}_0, [\mathsf{n}_0 \mapsto \mathsf{k}^{\mathsf{n}}_0]), (\mathsf{m}_0, \mathsf{k}^{\mathsf{m}}_0, [\mathsf{m}_0 \mapsto \mathsf{k}^{\mathsf{m}}_0]))$
Function $\mathsf{lca}(\pi_{\mathsf{n}}, \pi_{\mathsf{m}})$
    Input: A $<$-path $\pi_{\mathsf{n}} = \mathsf{n}_0, \ldots, \mathsf{n}$ ending in $\mathsf{n}$, represented by a
           tuple $(\mathsf{n}, \mathsf{k}^{\mathsf{n}}, \mathsf{KS}_{\mathsf{n}})$ where $\mathsf{KS}_{\mathsf{n}}$ is a map on the nodes $n$
           appearing in $\pi_{\mathsf{n}}$ s.t. $\mathsf{k}^{\mathsf{n}} = \max_{\mathsf{KS}_{\mathsf{n}}[\mathsf{n}]}$ and for any such $n$
             $\mathsf{KS}_{\mathsf{n}}[n] = \{ \mathsf{k}^{\mathsf{n}}_0 + \sum_i k_i \mid \mathsf{n}_0 <^{k_1} \ldots <^{k_c} n \text{ in } \pi_{\mathsf{n}} \}$
    Input: A $<$-path $\pi_{\mathsf{m}} = \mathsf{m}_0, \ldots, \mathsf{m}$ likewise
    if $\mathsf{k}^{\mathsf{n}} > \mathsf{k}^{\mathsf{m}}$ then return $\mathsf{lca}(\pi_{\mathsf{m}}, \pi_{\mathsf{n}})$
    if $\mathsf{n} \notin \pi_{\mathsf{m}}$ then switch $\mathsf{IDOM}[\mathsf{n}]$ do
        case $\perp$ do return $\perp$
        case $(\mathsf{n'}, \mathsf{k}^{\mathsf{n'}})$ do
            if $\mathsf{n'} \in \pi_{\mathsf{n}} \wedge \mathsf{n'} \notin \pi_{\mathsf{m}} \wedge |\mathsf{KS}_{\mathsf{m}}[\mathsf{m}]| > 1$ then return $\perp$
            $\mathsf{KS}_{\mathsf{n}}[\mathsf{n'}] \leftarrow \mathsf{KS}_{\mathsf{n}}[\mathsf{n'}] \cup \{\mathsf{k}^{\mathsf{n}} + \mathsf{k}^{\mathsf{n'}}\}$
            return $\mathsf{lca}\left(\left(\mathsf{n'}, \mathsf{k}^{\mathsf{n}} + \mathsf{k}^{\mathsf{n'}}, \mathsf{KS}_{\mathsf{n}}\right), \pi_{\mathsf{m}}\right)$
        end
    end

    if $\mathsf{k}^{\mathsf{n}} \in \mathsf{KS}_{\mathsf{m}}[\mathsf{n}]$ then return $(\mathsf{n}, \mathsf{k}^{\mathsf{n}})$
    if $\max_{\mathsf{KS}_{\mathsf{n}}[\mathsf{n}]} > \min_{\mathsf{KS}_{\mathsf{m}}[\mathsf{n}]} \wedge \min_{\mathsf{KS}_{\mathsf{n}}[\mathsf{n}]} < \max_{\mathsf{KS}_{\mathsf{m}}[\mathsf{n}]}$ then return $\perp$

    switch $\mathsf{IDOM}[\mathsf{n}]$ do
        case $\perp$ do return $\perp$
        case $(\mathsf{n'}, \mathsf{k}^{\mathsf{n'}})$ do
            $\mathsf{KS}_{\mathsf{n}}[\mathsf{n'}] \leftarrow \mathsf{KS}_{\mathsf{n}}[\mathsf{n'}] \cup \{\mathsf{k}^{\mathsf{n}} + \mathsf{k}^{\mathsf{n'}}\}$
            return $\mathsf{lca}\left(\left(\mathsf{n'}, \mathsf{k}^{\mathsf{n}} + \mathsf{k}^{\mathsf{n'}}, \mathsf{KS}_{\mathsf{n}}\right), \pi_{\mathsf{m}}\right)$
        end
    end
end
```

Algorithm 9: A timing sensitive *least common ancestor* algorithm.

9.4.2 An Algorithm for F

My algorithm for the computation of F from $<_{\text{TIME}}$ is based on the following fixed point characterization of F.

Observation 9.4.1. Let $G = (N, E)$ be any CFG, and F be the rule system

$$\frac{}{\mathsf{F}(n) \geq 0}\,\mathsf{F}^{\text{lin}}$$

$$\frac{\begin{array}{c} m \sqsupseteq^{k_m}_{\text{TIME}} n \qquad \mathsf{F}(n) \geq k_m \\ m' \sqsupseteq^{k}_{\text{TIME}} m \qquad k = \min\{\, k \mid m' \sqsupseteq^{k}_{\text{TIME}} m \,\} \qquad m' \neq n \\ \forall n \to_G x.\ m' \sqsupseteq^{k_m+k-1}_{\text{TIME}} x \ \wedge\ \mathsf{F}(x) \geq k_m + k - 1 \end{array}}{\mathsf{F}(n) \geq k_m + k}\,\mathsf{F}^{\text{trans}}$$

Then F $= \mu\mathsf{F}$ for the corresponding monotone functional F over the pointwise lattice on $N \to \mathbb{N}$.

The corresponding Algorithm 10 will

- remember for each node n with $\mathsf{F}(n) = k_m$ the current *witness* m such that $m \sqsupseteq^{k_m}_{\text{TIME}} n$.
- obtain the next mode $m' \neq n$ such that $m' \sqsupseteq^{k}_{\text{TIME}} m$ and $k = \min\{\, k \mid m' \sqsupseteq^{k}_{\text{TIME}} m \,\}$ simply by following one edge $m <^{k}_{\text{TIME}} m'$ in $<_{\text{TIME}}$.
- implement the check for successors x of n by looking for a cycle-free $<_{\text{TIME}}$-path from x to m' with cost $k_m + k - 1$.

Furthermore, by Observation 9.3.3, it is possible to replace the check $\mathsf{F}(x) \geq k_m + k - 1$ by only checking for fuel on "border" nodes[5] n' on the $<_{\text{TIME}}$-path from x to m', i.e.: to check

$$k_m + k - 1 - k_{n'} \leq \mathsf{F}\left(n'\right)$$

for nodes n' such that $n' \in N_M$ for some $<_{\text{TIME}}$-cycle M, where $k_{n'}$ is the amount of fuel spent when reaching node n'. But this means that *it is enough to compute the restrictions* $\mathrm{F}\big|_{N_{\mathbb{M}}}$ *of* F *to border nodes* $n' \in N_M$ for $<_{\text{TIME}}$-cycles M. Even if I am interested in the value F_n for *every node*, then these can be recovered from the restrictions $\mathrm{F}\big|_{N_{\mathbb{M}}}$ by Observation 9.3.3, and I do not need to compute them explicitly in Algorithm 10. In other words: Algorithm 10 only iterates over such border nodes $n \in N_{\mathbb{M}}$, where $\mathbb{M}$ is the set of $<_{\text{TIME}}$-cycles M, and

$$N_{\mathbb{M}} = \bigcup_{M \in \mathbb{M}} N_M$$

9.4.3 An Algorithm for $\mathrm{PDF}_{\sqsupseteq_{\text{TIME[FIRST]}}}$

Now that I have devised algorithms for $<_{\text{TIME}}$ and F, I am *almost* ready to use the characterization of $\mathrm{ipdom}_{\sqsupseteq_{\text{TIME}}}$ from Observation 9.3.4 and the rule system in Figure 9.5 on page 178 to give an algorithm for timing sensitive postdominance frontiers $\mathrm{PDF}_{\sqsupseteq_{\text{TIME[FIRST]}}}$.

I *could* use the rule system in Figure 9.5 *as is*, since the test

$$x \sqsupseteq_{\text{TIME[FIRST]}} y' \text{ in rule } \mathrm{PDF}_2^{\text{up}}$$

can be answered by $<_{\text{TIME}}$ and F (using the very definition of F, i.e.: Observation 9.3.2). But this would result in an inefficient algorithm

[5] of which there is at most one on each cycle-free $<_{\text{TIME}}$-path

```
Input  : A CFG $G = (N, E)$
Input  : The labeled pseudo forest $<_{\text{TIME}}$ for $G$
Input  : The set $N_{\mathbb{M}}$ of “border” nodes $n'$ in $<_{\text{TIME}}$
Output: A map $\mathsf{F} : N_{\mathbb{M}} \to \mathbb{N} \times N$ s.t. for all $n \in N_{\mathbb{M}}$: $\mathsf{F}[n] = (\mathrm{F}_n, m)$
        for some “witness” $m$
for $n \in N_{\mathbb{M}}$ do
    $\mathsf{F}[n] \leftarrow (0, n)$
end
changed $\leftarrow$ true
while changed do
    changed $\leftarrow$ false
    for $(n,\ (k_m, m)) \in \mathsf{F},\ m <^{k}_{\text{TIME}} m',\ m' \neq n$ do
        valid$_{m'}$ $\leftarrow$ true
        for $n \to_G x$ do if valid$_{m'}$ then
            let $n_0 <^{k_0}_{\text{TIME}} \ldots <^{k_c}_{\text{TIME}} n_{c+1}$ be the cycle-free $<_{\text{TIME}}$-path
              where $n_0 = x$ and $n_{c+1} = m'$
            if $\sum k_i = k_m + k - 1$ then
                if $\exists n_j.\ n_j \in N_{\mathbb{M}}$ and $k_m + k - 1 - \sum_{i<j} k_i > \mathsf{F}[n_j]$ then
                    valid$_{m'}$ $\leftarrow$ false
                end
            end
            else
                valid$_{m'}$ $\leftarrow$ false
            end
        end

        if valid$_{m'}$ then
            $\mathsf{F}[n] \leftarrow (k_m + k, m')$
            changed $\leftarrow$ true
        end
    end
end
return F
```

Algorithm 10: An algorithm for the computation of F from $<_{\text{TIME}}$. Here, $y <^{*} x$ means: $x = y \ \vee\ y <^{k_1} \ldots <^{k_c} x$

whenever the number of nodes in $<_{\text{TIME}}$-cycles and hence: the number of nodes x, y such that both

$$x \in \text{ipdom}_{\sqsupseteq_{\text{TIME[FIRST]}}}(z) \text{ and } x \in \text{ipdom}_{\sqsupseteq_{\text{TIME[FIRST]}}}(z)$$

is large.

Taking a closer look at rule PDF_1^{up} and rule PDF_2^{up} (and still writing $\sqsupseteq$ for $\sqsupseteq_{\text{TIME[FIRST]}}$), I notice that

1. nodes y are propagated unconditionally from $\text{PDF}_{\sqsupseteq}(z)$ to $\text{PDF}_{\sqsupseteq}(x)$ by PDF_1^{up} *unless* both x and z are in some common $<_{\text{TIME}}$ cycle M.
2. if x, z are in some common $<_{\text{TIME}}$ cycle M, and the $y \in \text{PDF}_{\sqsupseteq}(z)$ *originated* from some application of rule $\text{PDF}^{\text{local}}$ under substitution $[y \mapsto y, x \mapsto y']$ (i.e.: I have: $\neg\ y' \in \text{ipdom}_{\sqsupseteq}(y)$ and $y \to_G y'$) such that also $y' \in M$, then $y' <_{\text{TIME}} x$ and hence (by Observation 9.3.3): $x \sqsupseteq_{\text{TIME[FIRST]}} y'$. But this means that I can propagate y from $\text{PDF}_{\sqsupseteq}(z)$ to $\text{PDF}_{\sqsupseteq}(x)$ by rule PDF_2^{up}.

The resulting algorithm uses a map $\mathsf{DF} : N \to N \hookrightarrow \mathsf{Bool}$, where $\hookrightarrow$ indicates a partial map, and upon completion, y is in the $\sqsupseteq_{\text{TIME[FIRST]}}$-postdominance frontier of z iff $\mathsf{D}[x]$ is defined for z:

$$z \in \text{PDF}_{\sqsupseteq_{\text{TIME[FIRST]}}}(x) \quad \Leftrightarrow \quad z \in \text{dom}(\mathsf{D}[x])$$

During computation (and writing $\sqsupseteq$ for $\sqsupseteq_{\text{TIME[FIRST]}}$)

$$\begin{aligned}
\mathsf{D}[z][y] = \textbf{false} &\text{ indicates } && y \in \text{PDF}_{\sqsupseteq}(z)\\
\mathsf{D}[z][y] = \textbf{true} &\text{ indicates } && y \in \text{PDF}_{\sqsupseteq}(z)\\
&\textbf{and} && y \in \text{PDF}_{\sqsupseteq}(x) \text{ for } x \in \text{ipdom}_{\sqsupseteq}(z)
\end{aligned}$$

In other words: $\mathsf{D}[z][y]$ indicates whether y is to be propagated.

Propagation follows $\text{ipdom}_{\sqsupseteq_{\text{TIME[FIRST]}}}$ and hence: $<_{\text{TIME}}$. But $<_{\text{TIME}}$ is a pseudo forest, so it can be iterated by starting in leafs, and continuing towards (and: within!) roots. This is either done explicitly, or implicitly by using a priority queue, with each node assigned a priority based on a reverse depth first search starting in the roots. The priority queue based Algorithm 11 even works for arbitrary node-numbering (although then not necessarily efficiently).

```
Input  : A CFG $G = (N, E)$
Input  : Any numbering $^{\#} : N \to \mathbb{N}$
Input  : Immediate postdominators $\mathrm{ipdom}_{\sqsupseteq} = \mathrm{ipdom}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$
Input  : The set $N_{\mathbb{M}}$ of “border” nodes in $<_{\mathrm{TIME}}$
Data: A priority queue Q ordered by the numbering $^{\#}$
Output: $\mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$ represented as a map $\mathsf{DF} : N \to N \hookrightarrow \mathsf{Bool}$
$\mathsf{Q} \leftarrow \emptyset$
for $x \in N,\ y \to_G x,\ \neg\, x \in \mathrm{ipdom}_{\sqsupseteq}(y)$ do
    $\mathsf{DF}[x][y] \leftarrow$ true
    $\mathsf{Q} \leftarrow \mathsf{Q} \cup \{x\}$
end
while $\mathsf{Q} \neq \emptyset$ do
    $z \leftarrow \mathrm{remove}(\mathsf{Q})$ s.t. $z^{\#} = \max_{z \in Q} z^{\#}$
    for $x \in \mathrm{ipdom}_{\sqsupseteq}(z),\ (y, \mathbf{true}) \in \mathsf{DF}[z],\ \neg\, x \in \mathrm{ipdom}_{\sqsupseteq}(y)$ do
        $\mathsf{DF}_{x,y} \leftarrow \mathsf{DF}[x][y]$                    (may be $\perp$)
        $\mathsf{DF}'_{x,y} \leftarrow \mathsf{DF}_{x,y} \lor (z \notin N_{\mathbb{M}})$
        assert $\mathsf{DF}'_{x,y} \neq \perp$
        if $\mathsf{DF}'_{x,y} \neq \mathsf{DF}_{x,y}$ then
            $\mathsf{DF}[x][y] \leftarrow \mathsf{DF}'_{x,y}$
            $\mathsf{Q} \leftarrow \mathsf{Q} \cup \{x\}$
        end
    end
end
return DF
```

Algorithm 11: Computation of $\mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$

9.5 Soundness and Minimality of Timing Sensitive Control Dependence

I finish this chapter by making the empirical observation that timing sensitive empirical control dependence $\rightarrow_{\text{tscd}}$ is both sound and minimal with regard to *clocked* traces. Given any sequence

$$t = x_0,\ x_1,\ x_2,\ \ldots$$

its clocked sequence is just

$$t^{\text{⏱}} = x_0\text{⏱}\,[0]\,,\ x_1\text{⏱}\,[1]\,,\ x_2\text{⏱}\,[2]\,,\ \ldots$$

For clocked traces $t^{\text{⏱}}$ (i.e.: sequences of clocked (pseudo)-edges in G), the S-observable clocked trace $t^{\text{⏱}}\big|_S$ is defined by removing occurrences of unobservable clocked (partial) edges in the obvious way (similar to Definition 6.3.4 on page 80).

Observation 9.5.1 (Soundness of $\rightarrow_{\text{tscd}}$)**.** Let G be any CFG, and $M \subseteq N$ a set of nodes (the slicing criterion). Let $S = (\rightarrow_{\text{tscd}})^*\,(M)$ be the timing sensitive backward slice w.r.t M. Then, for any inputs i, i' such that

$$i \sim_S i'$$

I have

$$t_i^{\text{⏱}}\big|_S = t_{i'}^{\text{⏱}}\big|_S$$

Observation 9.5.2 (Minimality of $\rightarrow_{\text{tscd}}$)**.** Let G be any CFG, and $M \subseteq N$ a set of nodes (the slicing criterion). Let $S = (\rightarrow_{\text{tscd}})^*\,(M)$ be the timing sensitive backward slice w.r.t M. Then, for any $n \in S, n \notin M$ and $S' = S \setminus \{n\}$, there exist inputs i, i' such that

$$i \sim_{S'} i'$$

but:

$$t_i^{\text{⏱}}\big|_{S'} \neq t_{i'}^{\text{⏱}}\big|_{S'}$$

9.6 Timing Sensitive Control Dependence in Graphs with Unique Exit Node

My algorithms for timing sensitive control dependence $\to_{\text{tscd}}$ in arbitrary graphs from Section 9.4 are much more complicated than the corresponding algorithms for the computation of nontermination sensitive control dependence $\to_{\text{ntscd}}$. For $\to_{\text{ntscd}}$, I just needed Algorithm 5 (page 50), and then could directly use Algorithm 1 (page 28). For $\to_{\text{tscd}}$, on the other hand, I needed not only Algorithm 8 and the more complicated variant Algorithm 11 of Algorithm 1, but also Algorithm 10 for the computation of F.

The reason was that in graphs that are irreducible and lack a unique exit node n_x, the relation $<_{\text{TIME[FIRST]}}$ may not be transitive. There, only the relation $<_{\text{TIME}}$ is guaranteed to be transitive. But what about other graphs? For graphs *with* unique exit node n_x, or reducible graphs, Algorithm 10 is indeed not necessary, and the original Algorithm 1 is adequate for the computation of $\to_{\text{tscd}}$.

This is true simply due to the following observations:

Observation 9.6.1. Let G be a CFG with unique exit node n_x. Then

$$m \sqsupseteq_{\text{TIME}} n \iff m \sqsupseteq_{\text{TIME[FIRST]}} n$$

Also (writing $\sqsupseteq$ for $\sqsupseteq_{\text{TIME[FIRST]}}$): $\text{ipdom}^*_{\sqsupseteq} = \sqsupseteq$

Observation 9.6.2. Let G be a reducible CFG. Then

$$m \sqsupseteq_{\text{TIME}} n \iff m \sqsupseteq_{\text{TIME[FIRST]}} n$$

Also (writing $\sqsupseteq$ for $\sqsupseteq_{\text{TIME[FIRST]}}$): $\text{ipdom}^*_{\sqsupseteq} = \sqsupseteq$

Then by Observation 9.2.2, Observation 9.2.4 and Observation 9.2.5, $\sqsupseteq_{\mathrm{TIME[FIRST]}}$ admits an efficient PDF partitioning (Definition 3.2.7 on page 25), and I can use Algorithm 1 on any transitive reduction

$$<_{\mathrm{TIME[FIRST]}} = <_{\mathrm{TIME}}$$

of

$$\sqsupseteq_{\mathrm{TIME[FIRST]}} = \sqsupseteq_{\mathrm{TIME}}$$

But such a reduction is computed by Algorithm 11.

Summary

- Timing sensitive control dependence $\rightarrow_{\mathrm{tscd}}$ can be obtained by a modification of nontermination sensitive control dependence $\rightarrow_{\mathrm{ntscd}}$.
- No timing sensitive analogue of nontermination sensitive order dependence is necessary, not even in graphs without unique exit node.
- An algorithm for timing sensitive control dependency can be obtained by a modification of algorithms for nontermination sensitive control dependence.
- This algorithm is complicated for graphs without unique exit nodes, but for graphs with unique exit node a simple algorithm is sufficient.

10 Timing Dependence

As far as I'm concerned, if something is so complicated that you can't explain it in 10 seconds, then it's probably not worth knowing anyway.

(Bill Watterson — The Indispensable Calvin and Hobbes)

Timing sensitive control dependence $n \rightarrow_{\text{tscd}} m$ conflates two kind of dependencies:

- Decisions made at node n that decide whether node m is executed *at all* (i.e.: nontermination sensitive control dependence)
- Decisions made at node n that decide *when* node m is executed (i.e.: timing dependence).

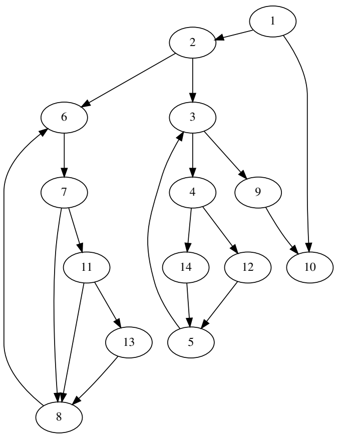
Consider again the graph G repeated in Figure 10.1a. For example, nodes 10 and 9 are both *nontermination sensitively control dependent* on node 3 (since by always successor node 4, node 3 can prevent them from ever executing). They are also *timing dependent* on the same node 3 (since *when* node 9 and 10 are executed is determined by how often node 3 chooses successor node 4 before eventually choosing successor node 9).

In contrast, node 5 is *nontermination sensitively control dependent* on node 3, but *not* timing dependent on node 3, since whenever node 5 executes after node 3, exactly three units of time will have passed.

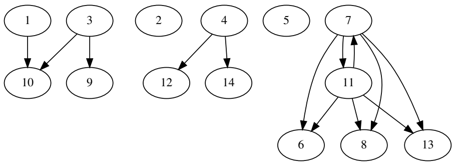
Also, node 8 is timing dependent on node 7, but *not* nontermination sensitively control dependent on node 7.

In this chapter, I propose a new notion $\rightarrow_{\text{td}}$ of *timing dependence* that isolates the latter kind of dependence. By a reduction to timing sensitive postdominance $\sqsupseteq_{\text{TIME[FIRST]}}$, this chapter's slogan will be

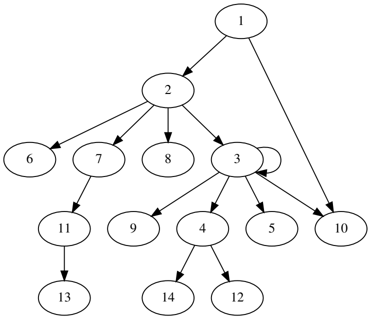
$$\rightarrow_{\text{td}} \quad \approx \quad |N| \times <_{\text{TIME[FIRST]}}$$



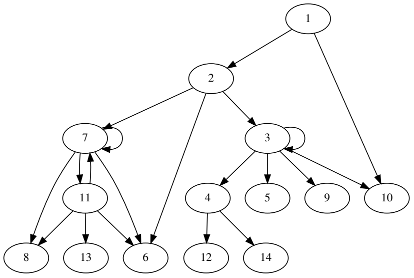
(a) A CFG



(b) Timing Dependence $\rightarrow_{\text{td}}$ for the CFG



(c) Its non-termination sensitive control dependence $\rightarrow_{\text{ntscd}}$



(d) Its timing sensitive control dependence $\rightarrow_{\text{tscd}}$

Figure 10.1: Timing Dependence

which expresses that timing dependence $\rightarrow_{\text{td}}$ in a graph $G = (N, E)$ can be reduced to the computation of $\sqsupseteq^{G_m}_{\text{TIME[FIRST]}}$ (represented by its transitive reduction $<^{G_m}_{\text{TIME[FIRST]}}$) for some graph G_m derived from G, for each node $m \in N$.

10.1 Timing Dependence

Definition 10.1.1. Let G be any CFG, and n, n', m nodes in G. I say that n' timing-sensitively postdominates n at position $k \in \mathbb{N}$ *on paths towards* m, and write

$$n' \sqsupseteq^{k \text{ in } G}_{\text{TIME[FIRST tow. } m]} n$$

iff on all G-paths starting in n that contain node m, n' first appears at time k. I omit "in G" whenever possible, and write

$$n' \sqsupseteq^{\text{ in } G}_{\text{TIME[FIRST tow. } m]} n$$

if this is the case for *some* k. Formally:

$$n' \sqsupseteq^{k \text{ in } G}_{\text{TIME[FIRST tow. } m]} n \quad \Leftrightarrow \quad \forall \pi \in {}_n\Pi^G.\, m \in \pi \;\rightarrow\; n' \in^k_{\text{FIRST}} \pi$$

$$n' \sqsupseteq^{\text{ in } G}_{\text{TIME[FIRST tow. } m]} n \quad \Leftrightarrow \quad n' \sqsupseteq^{k \text{ in } G}_{\text{TIME[FIRST tow. } m]} n \text{ for some } k \in \mathbb{N}$$

Proposition 10.1.1. Let G be any CFG, and n, n', m nodes in G. Then if m is unreachable in G from n, I have

$$n' \sqsupseteq^{k \text{ in } G}_{\text{TIME[FIRST tow. } m]} n \text{ for all } k \in N$$

If, on the other hand, m is reachable from n in G, let

$$G_m \;=\; G^{\rightarrow^* m}_{m \not\rightarrow}$$

be the graph obtained from G by removing all nodes that cannot reach m, and deleting all outgoing edges of m. Then n' first appears at time k in all maximal G-paths starting in n *that contain node* m iff n' first appears at time k in all maximal G_m-paths starting in n, i.e.:

$$n' \sqsupseteq^{k \text{ in } G}_{\text{TIME[FIRST tow. } m]} n \quad \Leftrightarrow \quad n' \sqsupseteq^{k \text{ in } G_m}_{\text{TIME[FIRST]}} n$$

To understand the upcoming definition of timing dependence $\rightarrow_{\text{td}}$, imagine for a conditional node n that for all G-successors x, x' of n I have

$$\begin{aligned} & n' \sqsupseteq^{k \text{ in } G}_{\text{TIME[FIRST tow. } m]} x \\ \text{and} \quad & n' \sqsupseteq^{k \text{ in } G}_{\text{TIME[FIRST tow. } m]} x' \end{aligned}$$

for *some* other node n'.

Then even if the timing of a node m after the execution of n is not constant, i.e.: even if I have

$$\neg\; m \sqsupseteq^{G}_{\text{TIME[FIRST]}} n$$

still any decision made at n cannot possibly *influence* the timing of m, since the timing of the other node n' after n *is* constant on paths towards m, *and* I always must reach n' before m. Hence any influence on the timing of m must happen at or after n'.

Definition 10.1.2. Let $G = (N, E)$ be any graph, and $m \neq n \in N$ be two nodes. Then m is *timing dependent* on n, and I write

$$n \rightarrow^{G}_{\text{td}} m$$

if there exists no node n' such that for *some* $k \in \mathbb{N}$ and *all* G-successors x of n, on all maximal G-paths starting in x *that contain node* m, n' first appears at time k, i.e.:

$$n \rightarrow^{G}_{\text{td}} m \iff \neg\; \exists n' \in N, k \in \mathbb{N}.\; \forall n \rightarrow_G x.\; n' \sqsupseteq^{k \text{ in } G}_{\text{TIME[FIRST tow. } m]} x$$

Observation 10.1.1. Timing dependence $\rightarrow_{\text{td}}$ is in fact the timing sensitive part of timing-sensitive control dependence $\rightarrow_{\text{tscd}}$, i.e.: for all graphs G and sets of nodes M,

$$(\rightarrow_{\text{td}} \cup \rightarrow_{\text{ntscd}})^{*}(M) \;=\; (\rightarrow_{\text{tscd}})^{*}(M)$$

10.2 Computation of Timing Dependence

Note that for every $m \in N$, the graph

$$G_m \;=\; G^{\rightarrow^* m}_{m \not\rightarrow}$$

is a graph with unique exit node m, and hence:

$$\sqsupseteq^{G_m}_{\text{TIME}} = \sqsupseteq^{G_m}_{\text{TIME[FIRST]}}$$

An Algorithm for $\rightarrow_{\text{td}}$ is available directly from the Algorithm 8 for $<_{\text{TIME}}$ on page 190, Proposition 10.1.1, and the following observation:

Observation 10.2.1. Let $G = (N, E)$ be any graph, and $m \neq n \in N$ be two nodes. Then

$$\begin{aligned} n \rightarrow^{G}_{\text{td}} m \quad &\iff \quad \neg\, \exists n' \neq n.\; n' \sqsupseteq^{G_m}_{\text{TIME[FIRST]}} n \\ &\iff \quad \neg\, \exists n' \quad\quad .\; n <^{G_m}_{\text{TIME}} n' \end{aligned}$$

for any transitive reduction $<^{G_m}_{\text{TIME}}$ of $\sqsupseteq^{G_m}_{\text{TIME}}$.

By this observation, I require a computation of $<^{G_m}_{\text{TIME}}$ for every node $m \in N$. Informally:

$$\rightarrow_{\text{td}} \quad \approx \quad |N| \;\times\; <_{\text{TIME[FIRST]}}$$

Summary

- In timing sensitive control dependence $\rightarrow_{\text{tscd}}$, timing and control dependence are entangled.
- “Pure” timing dependence $\rightarrow_{\text{td}}$ can be obtained from timing sensitive control dependence $\rightarrow_{\text{tscd}}$.

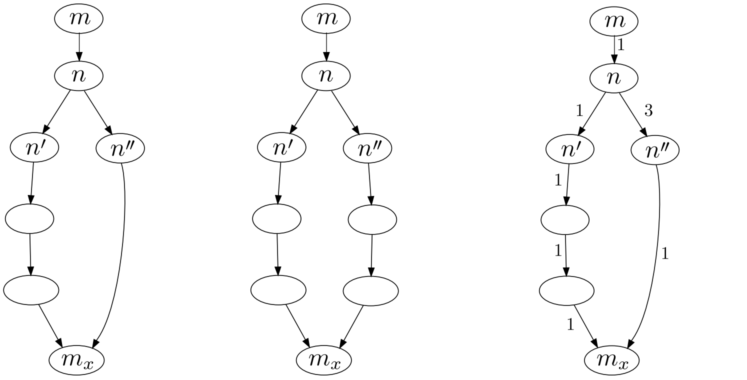
11 Timing Stratification

Time, Dr. Freeman? Is it really that time again? It seems as if you only just arrived.

(Valve — G-Man, Half-Life 2)

The two CFG in Figure 9.1 — repeated in Figure 11.1a and 11.1b — were the canonical example of CFG with and with and without timing leak. There, I assumed the set $S = \{m, m_x\}$ of observable nodes, and that the traversal of each CFG edge $n \rightarrow_G m$ took a unit amount $1u$ of time. The second CFG G' can be thought of as a *stratification* of the first CFG G: G' is obtained from G by adding two additional dummy nodes (and corresponding edges) "on the right"[1].

[1] and if i was dealing with edge-labeled CFG, I would label the new edges with an appropriate form of no-op



(a) A CFG G (b) A CFG G′ (c) Timing Cost C' for G

Figure 11.1: Dependence of execution time of m_x on n.

The same effect can be achieved by dropping the assumption that every edge in the CFG has a uniform execution time 1, and instead assume a timing *cost model* C, i.e.: a function

$$C : E \to \mathbb{N}^{+}$$

mapping each edge $(u, v) \in E$ of a CFG $G = (N, E)$ to a strictly positive natural number $C(u, v) > 0$: the amount of time spent traversing the edge $u \to_G v$. For example, I can *stratify* the CFG G in Figure 11.1a by using the timing cost model

$$C'(u, v) = \begin{cases} 3 & \text{for } (u, v) = (n, n'') \\ 1 & \text{otherwise} \end{cases} = \mathbb{1}\left[(n, n'') \mapsto 3\right]$$

shown in Figure 11.1c. Here, i assumed the initial timing cost model

$$C = \mathbb{1} := \ (u, v) \mapsto 1$$

Given an initial timing cost model C for some CFG G, a cost model $C' \geq C$ can be interpreted as a request for a compiler to insert appropriate additional nodes and "no-op"-edges to obtain a CFG G' as in the example, and otherwise keep the initial cost model C. However, making the modified C' explicit has the advantage of leaving the CFG unmodified, making the development in this chapter much simpler.

In this chapter, I will demonstrate how any "common" CFG— i.e.: any CFG that has a unique exit node n_x, or otherwise at least is reducible — can be *stratified*, in a precise sense which will justify the informal slogan

$$\text{Stratification + Timing Sensitivity} \ = \ \text{Nontermination Sensitivity}$$

This stratification will be "global" in the sense that it will stratify the timing at every conditional node n, independent of whether the choice at n is relevant w.r.t. a given set S of observable nodes. In other words: it will introduce some unnecessary (given some specific S) increase of timing cost. Despite these two restrictions, this chapter 11 will serve as the foundation of the application in chapter 12, in which I will demonstrate how to transform out timing leaks in *arbitrary* CFG, and also avoid unnecessary increase of timing cost.

11.1 Timing Sensitive Control Dependence for Arbitrary CFG with Cost Model

It is straightforward to extend the relevant definitions and algorithms for timing sensitive control dependence $n \to^{G}_{\text{tscd}} m$ of CFG G with the implicit timing cost model $\mathbb{1}$, to timing sensitive control dependence $n \to^{G[C]}_{\text{tscd}} m$ of CFG G with explicit timing cost model C. Essentially, I merely need to replace (implicit) occurrences of the cost "1" of some edge $n \to_G m$ with "$C\,(n, m)$".

Definition 11.1.1 (Generalization of Definition 9.1.1 on page 167)**.** Let G be any CFG, C a timing cost model for G, and n, m any nodes in G. Given any path

$$\pi = m_0, m_1, m_2, \ldots$$

I say that m appears in π at time k iff $m = m_i$ and

$$k = \sum_{0 \leq j < i} C\left(m_j, m_{j+1}\right)$$

In this case, I write $m \in^{k[C]} \pi$.

If additionally, $m_i \neq m$ for all $j < i$, i say that m *first* appears in π at time k, and write $m \in^{k[C]}_{\text{FIRST}} \pi$.

Furthermore, I say that m $\sqsupseteq^{k[C]}_{\text{TIME[FIRST]}}$-postdominates n in G iff on all maximal G-paths starting in n, m first appears at time k. I omit "in G" whenever possible, and say that m $\sqsupseteq^{C}_{\text{TIME[FIRST]}}$-post dominates n iff m $\sqsupseteq^{k[C]}_{\text{TIME[FIRST]}}$-post dominates n for some $k \in \mathbb{N}$. Formally:

$$\begin{aligned}
m \sqsupseteq^{k[C] \text{ in } G}_{\text{TIME[FIRST]}} n \quad &\Leftrightarrow \quad \forall \pi \in {}_n\Pi^{G}_{\text{MAX}}.\ m \in^{k[C]}_{\text{FIRST}} \pi \\
m \sqsupseteq^{C \text{ in } G}_{\text{TIME[FIRST]}} n \quad &\Leftrightarrow \quad \forall \pi \in {}_n\Pi^{G}_{\text{MAX}}.\ m \in^{k[C]}_{\text{FIRST}} \pi \text{ for some } k \in \mathbb{N}
\end{aligned}$$

Remark 11.1.1. Obviously, Definition 11.1.1 really *is* a generalization of Definition 9.1.1, i.e.:

$$m \sqsupseteq^{k}_{\text{TIME[FIRST]}} n \quad \Longleftrightarrow \quad m \sqsupseteq^{k[\mathbb{1}]}_{\text{TIME[FIRST]}} n$$

I define Timing sensitive control dependence for a CFG G with timing cost model C just as expected, but I must not forget to account for the timing cost of traversing an edge $n \rightarrow_G n_l$ or $n \rightarrow_G n_r$ in G.

Definition 11.1.2 (Generalization of Definition 9.1.2 on page 168)**.** Let G be any CFG, C a timing cost model for G, and n, m any nodes in G. Then m is said to be *timing sensitively control-dependent* on n under cost model C, written $n \rightarrow^{G[C]}_{\text{tscd}} m$ or just $n \rightarrow^{C}_{\text{tscd}} m$, if there exists G successors n_l and n_r of n, and some $k \in \mathbb{N}$ such that for the unique k_l, k_r that satisfy

$$k = k_l + C\,(n, n_l) \quad \text{and} \quad k = k_r + C\,(n, n_r)$$

m $\sqsupseteq^{k_l[C]}_{\text{TIME[FIRST]}}$-post dominates n_l, but *not*: m $\sqsupseteq^{k_r[C]}_{\text{TIME[FIRST]}}$-post dominates n_r.

Formally: $n \rightarrow^{C}_{\text{tscd}} m \Leftrightarrow$

$$\begin{aligned} & m \sqsupseteq^{k_l[C]}_{\text{TIME[FIRST]}} n_l \quad \text{and} \\ \neg \quad & m \sqsupseteq^{k_r[C]}_{\text{TIME[FIRST]}} n_r \end{aligned}$$

for some $k \in \mathbb{N}$ and n_l, n_r such that $n \rightarrow_G n_l$ and $n \rightarrow_G n_r$, where k_l, k_r are defined (given k, n_l, n_r) as above.

Remark 11.1.2. Obviously, Definition 11.1.2 really *is* a generalization of Definition 9.1.2, i.e.:

$$n \rightarrow_{\text{tscd}} m \quad \Longleftrightarrow \quad n \rightarrow^{\mathbb{1}}_{\text{tscd}} m$$

With the example of this generalization it still requires some care, but is otherwise routine to generalize all remaining notions algorithms from Chapter 9 to CFG and arbitrary timing cost model C. Specifically, not only can I generalize definitions

$$\begin{array}{lrcl} & m \rightarrow_{\text{tscd}} n & \text{to} & n \rightarrow^{C}_{\text{tscd}} m \\ \text{and} & m \sqsupseteq^{k}_{\text{TIME[FIRST]}} n & \text{to} & m \sqsupseteq^{k[C]}_{\text{TIME[FIRST]}} n \\ \\ \text{but also:} & \mathsf{T}_{\text{FIRST}} & \text{to} & \mathsf{T}^{C}_{\text{FIRST}} \\ & m \sqsupseteq^{k}_{\text{TIME}} n & \text{to} & m \sqsupseteq^{k[C]}_{\text{TIME}} n \\ & \mathsf{T} & \text{to} & \mathsf{T}^{C} \\ & \mathrm{F}_n & \text{to} & \mathrm{F}^{C}_{n} \\ & \mathsf{F} & \text{to} & \mathsf{F}^{C} \end{array}$$

as well as algorithms 10 to F^{C}_{n} and 8 to $<^{C}_{\text{TIME}}$.

I can also generalize Algorithm 11 to $\text{PDF}_{\sqsupseteq^{C}_{\text{TIME}}}$, but *this* generalization is not completely straight-forward. I describe it in Appendix E.

11.2 Timing-Stratification

Given a CFG G and a timing cost model C for G, to what extend can I hope to stratify the timing behavior of G? Ideally, I would want all (otherwise) observably equivalent executions of G to also take the same amount of time, but this is clearly impossible in the presence of (unobservable) *loops* in G. The best I can hope to do is to find a modified cost model $C' \geq C$ such that for any observably equivalent executions *in which all loops execute the same number of times*, the executions also take the same total amount of time. But I already have appropriate notions to characterize this difference:

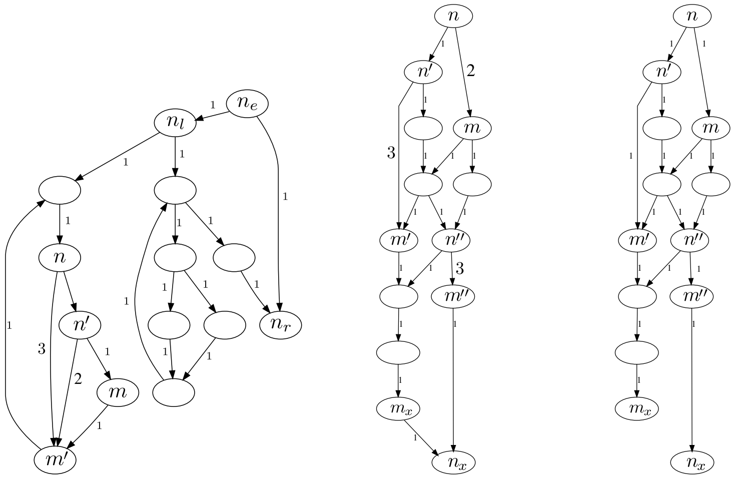
- Timing-sensitive postdominance $\sqsupseteq_{\text{TIME[FIRST]}}$ and timing sensitive control-dependence $\rightarrow_{\text{tscd}}$ are sensitive to *all* timing.
- Nontermination sensitive postdominance $\sqsupseteq_{\text{MAX}}$ and nontermination sensitive control-dependence[2] $\rightarrow_{\text{ntscd}}$ are sensitive to possible non-termination (hence: sensitive to the number of times a loop may execute), but otherwise *insensitive* to timing.

Definition 11.2.1. Let G be any CFG, and C' a timing cost model for G. Then i say that the timing of G under C' is *stratified* if for all sets $M \subseteq N$ of nodes, the timing sensitive backwards slice of M under C' is equal to the nontermination sensitive backward slice of M, i.e.:

$$\left(\rightarrow_{\text{ntscd}} \ \dot\cup \ \rightarrow_{\text{ntsod}}\right)^* \quad = \quad \left(\rightarrow^{C'}_{\text{tscd}}\right)^*$$

The timing of all CFG in Figure 11.2 is stratified under their respective cost model as shown. Note that in Figure 11.2a, I do *not* need to delay the edge $n_e \rightarrow_G n_r$ (for example, by setting $C(n_e, n_r) = 4$), because I already have $n_e \rightarrow_{\text{ntscd}} n_r$ due to the fact that continuing from n_l, the node n_r not necessarily has to be executed. On the other (left) hand, the delays at n and n' stratify this "**if** … **then if** …" region.

[2] and decisive order dependence $\rightarrow_{\text{dod}} = \rightarrow_{\text{ntsod}}$



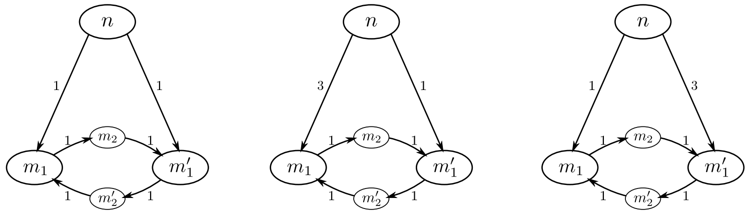
(a) A CFG (b) A CFG G (c) A CFG G'

Figure 11.2: Some CFG with stratified timing cost model

In Figure 11.2b, the delay $C\,(n', m') = 3$ at edge $n' \rightarrow_G m'$ is necessary, but *not* in order to attempt to establish $m' \sqsupseteq^{C,3}_{\text{TIME[FIRST]}} n'$, which does *not* hold (and *cannot* hold for any C or any k, because $\neg\ m' \sqsupseteq_{\text{MAX}} n'$). Instead, this delay is necessary to establish $n_x \sqsupseteq^{C,k}_{\text{TIME[FIRST]}} n'$ for $k = 7$ and the unique exit node n_x, which I must establish (for some k), since $n_x \sqsupseteq_{\text{MAX}} n'$. From an information flow point of view, I must establish this and the other two delays because the execution (time) of n_x might be deemed observable.

On the other hand in Figure 11.2c, which is obtained from Figure 11.2b by removing the edge $m_x \rightarrow_G n_x$, already the default timing cost model $\mathbb{1}$ stratifies the CFG. This example is crucial for information flow control, since removal of this edge certainly does *not* change any observation if n_x is *un*observable[3]. In this case, I do *not* need to delay

[3] except for the fact that one will observe the pseudo-edge $(m_x, \bot)$ instead of (m_x, n_x), but this does *not* inform an observer about the *input* to a given execution



(a) $\mathbb{1}$ for G (b) C for G (c) C' for G

Figure 11.3: Impossibility of Stratification of a CFG G

any edges! I will make use of this technique (removing certain edges, depending on a given set of *observable* nodes) in Chapter 12.

Can I expect to find a stratification C' for every arbitrary CFG (in the sense of Definition 11.2.1)? Consider the CFG in Figure 11.3. It is easy to see that n controls the order *neither* of m_1, m_2 nor of m'_1, m'_2, since, for example, m_1 *always* occurs before m_2 (when starting in n). So for $M = \{m_1, m_2\}$ and $M = \{m'_1, m'_2\}$ i have

$$\begin{array}{rrcl} & n \notin \ (\rightarrow_{\text{ntscd}} \ \dot\cup \ \rightarrow_{\text{ntsod}})^* (M) & = & M \\ \text{and} & n \notin \ (\rightarrow_{\text{ntscd}} \ \dot\cup \ \rightarrow_{\text{ntsod}})^* (M') & = & M' \end{array}$$

On the other hand, I have both $n \rightarrow_{\text{tscd}} m_1$ and $n \rightarrow_{\text{tscd}} m'_1$, so under the default timing cost model $\mathbb{1}$:

$$\begin{array}{rrcl} & n \in \ \left(\rightarrow^{\mathbb{1}}_{\text{tscd}}\right)^* (M) & = & M \cup \{n\} \\ \text{and} & n \in \ \left(\rightarrow^{\mathbb{1}}_{\text{tscd}}\right)^* (M') & = & M' \cup \{n\} \end{array}$$

Can I find *some* timing cost model to stratify G? I can certainly find a cost model C such that $\neg\ n \rightarrow^{C}_{\text{tscd}} m_1$ (as shown in Figure 11.3b). I can also find a different cost model C such that $\neg\ n \rightarrow^{C}_{\text{tscd}} m'_1$ (Figure 11.3c).

But I cannot find a cost model C' such that both hold at the same time, for this would require there to be some $k = C'(n, m_1)$ such that $n \sqsupseteq^{C',k}_{\text{TIME[FIRST]}} m_1$, *and* some $k' = C'\left(n, m'_1\right)$ with $n \sqsupseteq^{C',k'}_{\text{TIME[FIRST]}} m'_1$. In other words, a cost model C' would need to simultaneously solve the two following equations:

$$\begin{array}{rcccccccl} C'\left(n, m'_1\right) & + & C'\left(m'_1, m'_2\right) & + & C'\left(m'_2, m_1\right) & - & C'\left(n, m_1\right) & = & 0 \\ -C'\left(n, m'_1\right) & + & C'\left(m_1, m_2\right) & + & C'\left(m_2, m'_1\right) & + & C'\left(n, m_1\right) & = & 0 \end{array}$$

But this is impossible, since all timing cost model are required to be *strictly* positive (i.e.: $C' > 0$).

11.3 An Algorithm for Timing Stratification

I have just shown that it is impossible to find a stratification for *every* CFG (and initial timing cost model C). Nevertheless, in this section I will show an algorithm that *will* return a stratification when applied to a CFG G in which no nodes are (indecisively, nontermination sensitively) order dependent, i.e.: $\rightarrow^{G}_{\text{ntsod}} = \emptyset$. Note that this class of CFG includes both of the following classes:

1. CFG that are reducible
2. CFG with unique exit node n_x

The key to this algorithm is the observation that I can always stratify such CFG by only delaying CFG-edges $n \rightarrow_G n'$ *at conditional nodes* $n \in \text{COND}$. Given such a node n, I will usually delay *some* but not all of its outgoing edges, going from $C(n, n')$ to $C'(n, n')$ such that $C' \geq C$. I urge the reader to confirm that all examples of stratified CFG shown so far follow this scheme!

The next observation is equally important: whenever timing sensitive post-dominance fails *only due to timing*, i.e.: whenever

$$\begin{array}{rrl} & & m \sqsupseteq_{\text{MAX}} n \\ \text{but} & \neg & m \sqsupseteq^{C}_{\text{TIME}} n \end{array}$$

then this can inductively reduced to a timing cost model $C' \geq C$ such that for some G-successors n_l, n_r of n I have

$$\begin{array}{rccc} & m \sqsupseteq^{k_l[C']}_{\text{TIME}} n_l & \text{and} & m \sqsupseteq^{k_r[C']}_{\text{TIME}} n_r \\ \text{but} & \underbrace{C'(n, n_l) + k_l}_{=:K_l} & < & \underbrace{C'(n, n_r) + k_r}_{=:K_r} \end{array}$$

Now the idea for my algorithm is obvious: in this case, just update C' at (n, n_l) to $K_r - k_l$

$$C'[(n, n_l) \mapsto K_r - k_l]$$

The resulting Algorithm 12 is similar to Algorithm 8. The Algorithm 20 (shown on page 398 in the appendix) that computes a least common ancestor of two successor nodes of some node x is only ever called if x has *some* nontermination sensitive immediate postdominator. It also computes a "correction" ΔC of the current timing model C' such that the common ancestor z returned is a timing sensitive least common ancestor *under the timing model* $C' + \Delta C$ (where addition is pointwise).

The computation in Algorithm 12 initializes the $\mathbb{N}$-labeled pseudo-forest $<$ with a transitive reduction $<^{C}_{\text{TIME}}$ [4]. It proceeds by updating $<$ and the timing cost model C' until finally, $<$ is (structurally) a transitive reduction of *maximal-path*[5] postdominance $\sqsupseteq_{\text{MAX}}$, and (except for self-edges) equal to the transitive reduction $<^{C'}_{\text{TIME}}$ under the *updated* cost model C'. The computation always terminates.

Observation 11.3.1. Let G be a CFG such that no nodes are (indecisively, nontermination sensitively) order dependent, i.e.: $\rightarrow^{G}_{\text{ntsod}} = \emptyset$. Let C be any timing cost model of G, and C' the timing cost model computed by Algorithm 12. Then $C' \geq C$, and the timing of G is stratified under C'.

Remember that for G be a CFG with unique exit node n_x, I have $\rightarrow^{G}_{\text{ntsod}} = \emptyset$. The same holds for reducible G.

[4] which can be obtained from Algorithm 26 on page 415, which itself is just a minor modification of Algorithm 8 on page 190

[5] i.e.: nontermination sensitive, but not timing sensitive

```
Input  : A CFG $G$
Input  : A initial timing cost model $C$
Data: A $\mathbb{N}$ labeled pseudo-forest $<$, represented as a map
      $\mathsf{IDOM} : N \hookrightarrow N \times \mathbb{N}$ s.t. $\mathsf{IDOM}[n] = (m, k)$ iff $n <^k m$
Output: A timing cost model $C' \geq C$
begin
    $C' \leftarrow C$
    $< \leftarrow <^C_{\mathrm{TIME}}$
    $\mathsf{workset} \leftarrow N_{\mathrm{COND}} \leftarrow \mathrm{COND}_G \cap \{ n \mid n <_{\mathrm{MAX}} m \text{ for some } m \}$
    while $\mathsf{workset} \neq \emptyset$ do
        $x \leftarrow remove(\mathsf{workset})$
        $\left(z, k, \boxed{\Delta C}\right) \leftarrow \mathsf{lca}_< (\{ (y, C'(x, y)) \mid x \rightarrow_G y \})$ via alg. 20
        if $(z, k, \Delta C) \neq \bot$ then
            if $(z, k) \neq \mathsf{IDOM}[x]$ then
                $\mathsf{IDOM}[x] \leftarrow (z, k)$
                $\mathsf{workset} \leftarrow$
                  $\mathsf{workset} \cup \{n \in N_{\mathrm{COND}} \mid n \neq x, \exists n \rightarrow_G y.\ y <^* x\}$
            end
            $C' \leftarrow C' + \Delta C$
        end
    end
    assert $\forall n \neq m.\ \forall k.\quad n <^k m \Leftrightarrow n <^{k[C']}_{\mathrm{TIME}} m$
    assert $\forall n \neq m.\quad n <^* m \Leftrightarrow m \sqsupseteq_{\mathrm{MAX}} n$
    return $C'$
end
```

Algorithm 12: An efficient algorithm for the computation of a timing stratification $C' \geq C$ of some CFG with initial timing cost model C. Here, $y <^* x$ means: $x = y \lor y <^{k_1} \ldots <^{k_c} x$

Summary

- Up to a difference in cost model, timing sensitive control dependence $\rightarrow_{\text{tscd}}$ and nontermination insensitive control dependence $\rightarrow_{\text{ntscd}}$ are essentially the same, in a large class of graphs.
- The timing model that witnesses this correspondence can be computed by a simple algorithm.

Part III

Timing Sensitive Software Security

Es ist nicht genug, zu wissen, man muß auch anwenden;
es ist nicht genug, zu wollen, man muß auch tun.

(Johann Wolfgang von Goethe — Wilhelm Meisters Wanderjahre)

12 Transforming Out Timing Leaks in Arbitrary CFG

If we could perceive time as it really was — what reason would grammar professors have to get out of bed?

(Rosalind and Robert Lutece — Bioshock Infinite)

When transforming out timing leaks, the general goal is — given a program *and* some form of policy that specifies limits on the allowed flow of information — to transform the program such that the transformed program is secure with regard to the given policy. In existing approaches (e.g.: in [Aga00; Mol+06; BRW06]) as well as in the approach I introduce in this chapter, this is only possible if the policy allows information flow due to the number of execution of the programs loops. For example in a simple $\{L, H\}$-lattice based policy: only if loop predicates whose execution time is L-observable do not depend on H input.

The canonical example of such an approach is [Aga00]. There, the policy consists of a mapping from *left-expressions* (which, aside from program variables, also consists of expressions obtained by dereferencing record-variables, or by indexing array-variables) to security levels L and H. The transformation is syntax-tree-directed, and specified relative to the result of an initial *timing insensitive* information flow analysis, which in turn is specified in form of a security type system. Specifically, the transformation will only successfully transform a loop **while** $e\ C$ if the type-system inferred the expression e to be of security type L, and *fail* otherwise. For branches **if** $e\ C_1$ **else** C_2, timing is harmonized by *cross-copying* parts of one branch into the other.

This somewhat analogous to my algorithm for the timing stratification of CFG from Chapter 11. Imagine a security policy in form of a mapping lvl that maps CFG-nodes to security levels L, H. I deem a program secure if the timing-sensitive backwards-slice of L nodes does not contain H nodes. Then timing stratification is

1. specified relative to the result of *timing insensitive* postdominators $\sqsupseteq_{\text{MAX}}$[1]
2. successful only up to nontermination sensitive control dependence $\rightarrow_{\text{ntscd}}$, because there is some node n with $\text{lvl}(n) = \text{H}$ in the timing-sensitive backwards slice of L nodes in the *transformed program*[2] if and only if such n is in the nontermination sensitive backward slice of L nodes.

The two major *differences* are:

1. Instead of transforming the program (i.e.: the CFG), timing stratification merely change the timing cost model C to $C' \geq C$. But as said before, this then can be understood as a directive to a compiler to insert additional **skip**/**no-op** instructions, according to the difference $C' - C$ of the stratified timing cost model and the original cost model.
2. Timing stratification is *independent* of the given information flow policy. Instead, it anticipates *all* possible policies, by attempting to harmonize timing-sensitive slicing with nontermination sensitive slicing for *all* slicing criteria M.

I will attack the second difference any minute now, by a simple CFG-preprocessing. But first, I remark with regard to the first difference that the approach [Aga00] and similar techniques (see, e.g., [MS15] for a comparison) have the advantage that they do not require an explicit initial timing cost model C. Indeed, such a model may be hard to get hold of for execution models with under-specified timing behavior (such as the Java Virtual Machine), or otherwise timing models that are at least hard to obtain statically. Just think, for example, of x86-machine code, and the effect of complex memory-cache hierarchies. The transformation in [Aga00] (and others) somewhat alleviate

[1] since in Algorithm 12, i only try so stratify conditional nodes x for which there exists a immediate $\sqsupseteq_{\text{MAX}}$-postdominator
[2] i.e.: under the resulting timing cost model C'

this problem by *cross-copying*[3] program segments, such that — if so required — both branches essentially execute the same code, and hence can be hoped to have the similar timing behavior (or even the same timing behavior, if the timing model is very simple). In contrast, my approach makes use of the timing cost model C to gauge the execution time of all branches, and equalize (*stratify*) their timing behavior artificially. I suspect — but did not try to establish empirically — that my approach can lead to less runtime overhead in the transformed programs. But again: my approach *does* require an initial timing model C.

Approaches in the tradition of [Aga00] are based on some form of *cross copying*: If a branch condition is H-dependent, code mimicking the behavior of the left branch is appended to the right branch, *and* code mimicking the behavior of the right branch is appended to the left branch. My approach instead is again based not on *cross-copying*, but on the modification of static timing cost model C. To me it appears very difficult to even attempt to apply *cross-copying* or similar techniques in *arbitrary* CFG, since these ideas explicitly rely on the fact that the program model (syntax tree) is *structured* by definition (specifically, it has no arbitrary jumps of control flow). Indeed note that the approach in [Wu+18] which works *not* on the syntax tree but on the CFG still requires the CFG to be of a particular structure. The authors do claim an algorithm to transform CFG into this form *by inserting new variables, and introducing new conditional nodes branching on these new variables*, but it is not clear to me whether their algorithm works for arbitrary CFG. Nor is clear to me the extend to which their method degenerates to the Böhm-Jacopini construction[BJ66] in the general case. Presumably, it is possible to directly apply *cross-copying* or similar techniques to *reducible* CFG which can be obtained by *node splitting*. In general, however this may lead to exponential growth [CFT03]. Though I did not attempt it, what *may* be possible is to apply *cross-copying* (or similar) techniques not to the CFG G, but to some

[3] or similar

CFG $G'(M)$ dependent on the slicing criterion M[4] that is “reducible wherever it matters”, and then apply the cross-copying result back to the original CFG G.

[4] for example: $G' = G_{M\not\to}$

12.1 An Naive Algorithm

My goal of this section is to give a first naive algorithm that — given an *arbitrary* CFG, an arbitrary timing cost model C, and a slicing criterion (i.e.: a set of nodes) M — finds a timing cost model $C' \geq C$ such that for this criterion M

$$(\rightarrow_{\text{ntscd}} \mathbin{\dot\cup} \rightarrow_{\text{ntsod}})^* (M) \quad = \quad \left(\rightarrow^{C'}_{\text{tscd}}\right)^* (M)$$

In that case I will say that C' transforms out timing leaks *relevant to M* of G under C *up to timing leaks due to loops*[5].

My algorithm for transforming out timing leaks is merely an application of the Algorithm 12 for timing stratification. It is based on observations somewhat similar to those that lead to the reduction of nontermination insensitive slicing with $\rightarrow_{\text{ntiod}}$ and $\rightarrow_{\text{nticd}}$ to nontermination insensitive slicing with just $\rightarrow_{\text{nticd}}$ in Section 7.4.

1. The Algorithm 12 for timing stratification only fails to compute a timing cost model C' such that the timing is stratified for *all* slicing criteria, if there is some $<_{\text{TIME}}$-cycle (remember the example from Figure 11.3 on page 215).
2. But such $<_{\text{TIME}}$-cycles (at least: those relevant to a given slicing criterion M) can be broken by *deleting in G* all outgoing edges of nodes $m \in M$.
3. Additionally, as exemplified in Figure 11.2c on page 214, deleting such edges will eliminate the need to delay (some) edges which during *stratification* were only delayed in anticipation of *arbitrary* slicing criteria M.

The transformation then simply works by *stratifying* not the original CFG G under the original timing cost model C, but instead the CFG $G_M := G_{M\not\rightarrow}$ obtained from G by removing all edges leaving M. The

[5] remember that I cannot hope to transform out timing leaks due to loops

stratification is done under the restriction $C_M = C|_{E_M}$ of the original cost model C to the edges E_M still present in G_M. The result is a timing cost model $C'_M \geq C_M$ for G_M, which can simply be extended to a timing cost model $C' \geq C$ for G by using the original timing cost $C(m, m')$ for edges $m \to_G m'$ missing in G_M.

Observation 12.1.1. Let $G = (N, E)$ be an *arbitrary* CFG, $M \subseteq N$ any set of nodes (the slicing criterion), and C any timing cost model of G.

Then for $G_M := (N, E_M) := G_{M\not\to}$ and $C_M = C|_{E_M}$, let $C'_M \geq C_M$ be the result of running Algorithm 12 on G_M under C_M, and

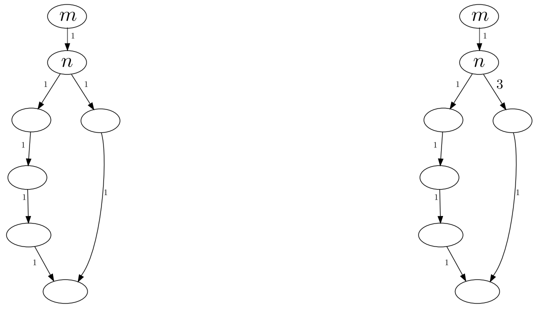
$$C'(n, m) := \begin{cases} C'_M(n, m) & \text{if } n \to_{G_M} m \\ C(n, m) & \text{otherwise} \end{cases}$$

Then C' transforms out timing leaks *relevant to* M of G under C *up to timing leaks due to loops*, i.e.:

$$\left(\to^{G}_{\text{ntscd}} \cup \to^{\text{G}}_{\text{ntsod}}\right)^{*}(M) \quad = \quad \left(\to^{G[C']}_{\text{tscd}}\right)^{*}(M)$$

Remark 12.1.1. Remember that by Observation 7.2.1, I also have

$$\left(\to^{G}_{\text{ntscd}} \cup \to^{\text{G}}_{\text{ntsod}}\right)^{*}(M) \quad = \quad \left(\to^{G_{M\not\to}}_{\text{ntscd}}\right)^{*}(M)$$



(a) A CFG without timing leak for $M = \{m\}$

(b) Removal of Timing Leaks via Observation 12.1.1

Figure 12.1: Dependence of execution time of m_x on n.

12.2 A More Precise Algorithm

The reduction from G to $G_M = G_{M\not\to}$ works in the sense of Observation 12.1.1, but introduces unnecessary delays. The simplest example is shown in Figure 12.1. There, the timing leaks transformation shown in Figure 12.1b makes an unnecessary (relative to $M = \{m\}$) delay at n. The transformation here delays an edge at the conditional node n *even though* the node m is not even timing dependent on n.

Such unnecessary delays are avoided by the following Algorithm 13. Apart from to the transition of G to G_M, it is merely a small modification to Algorithm 12. The crucial difference is highlighted. The algorithm only ever considers conditional nodes n such that n is in the timing sensitive backward slice of M, but not in the nontermination sensitive backward-slice of M. For the example in Figure 12.1a, which shows a CFG G with initial timing cost model $C = \mathbb{1}$, and slicing criterion $M = \{m\}$, Algorithm 13 returns the unchanged model $C' = C$.

Observation 12.2.1. Let $G = (N, E)$ be an *arbitrary* CFG, $M \subseteq N$ any set of nodes (the slicing criterion), and C any timing cost model of G.

```
**Input**  : A CFG $G = (N, E)$
**Input**  : A initial timing cost model $C$ for $G$
**Input**  : A slicing criterion $M$
**Data:** A $\mathbb{N}$ labeled pseudo-forest $<$, represented as a map
       $\mathsf{IDOM} : N \hookrightarrow N \times \mathbb{N}$ s.t. $\mathsf{IDOM}[n] = (m, k)$ iff $n <^k m$
**Output:** A timing cost model $C' \geq C$
**begin**
    $G_M \leftarrow (N, E_M) \leftarrow G_{M\not\rightarrow}$
    $C_M \leftarrow C|_{E_M}$
    $S \leftarrow \left(\to^{G_M[C_M]}_{\text{tscd}}\right)^*(M) \setminus \left(\to^{G_M}_{\text{ntscd}}\right)^*(M)$
    **assert** $S = \left(\to^{G[C]}_{\text{tscd}}\right)^*(M) \setminus \left(\to^{G}_{\text{ntscd}} \dot\cup \to^{G}_{\text{ntsod}}\right)^*(M)$
    $C'_M \leftarrow C_M$
    $< \;\leftarrow <^{G_M[C_M]}_{\text{TIME}}$
    $\mathsf{workset} \leftarrow N_{\text{COND}} \leftarrow \text{COND} \cap \{\, n \mid n <^{G_M}_{\text{MAX}} m$ for some $m \,\} \cap S$
    **while** $\mathsf{workset} \neq \emptyset$ **do**
        $x \leftarrow \mathit{remove}(\mathsf{workset})$
        $(z, k, \Delta C) \leftarrow \mathsf{lca}_{<}\left(\{\, (y, C'(x, y)) \mid x \to_{G_M} y \,\}\right)$ **via** alg. 20
        **if** $(z, k, \Delta C) \neq \bot$ **then**
            **if** $(z, k) \neq \mathsf{IDOM}[x]$ **then**
                $\mathsf{IDOM}[x] \leftarrow (z, k)$
                $\mathsf{workset} \leftarrow$
                  $\mathsf{workset} \cup \{n \in N_{\text{COND}} \mid n \neq x, \exists n \to_G y.\, y <^* x\}$
            **end**
            $C'_M \leftarrow C'_M + \Delta C$
        **end**
    **end**
    $C' \leftarrow C|_{E \setminus E_M} \cup C'_M$
    **return** $C'$
**end**
```

Algorithm 13: An efficient algorithm transforming out timing leaks, given a slicing criterion M for in some CFG with initial timing cost model C. Here, $y <^* x$ means: $x = y \lor y <^{k_1} \ldots <^{k_c} x$

Let $C' \geq C$ be the result of running Algorithm 13 on G under C. Then C' transforms out timing leaks *relevant to M* of G under C *up to timing leaks due to loops*, i.e.:

$$\left(\rightarrow^{G}_{\text{ntscd}} \;\dot\cup\; \rightarrow^{G}_{\text{ntsod}} \right)^{*}(M) \quad = \quad \left(\rightarrow^{G[C']}_{\text{tscd}} \right)^{*}(M)$$

The timing leaks transformation $C' \geq C$ obtained from running Algorithm 13 on G under C is never worse than the timing stratification C'' for $G_{M\not\to}$ obtained from Algorithm 12 along the lines of Observation 12.1.1.

Observation 12.2.2. Let C', C'' as above. Then $C' \leq C''$.

But is the timing leaks transformation C' in some global sense optimal? I cannot answer this question for the reason that I did not attempt formulate such a criterion! The only reason I know that Algorithm 13 is never worse than Algorithm 12 is that both only ever delay timing at conditional nodes n and their G-successors, and hence both limit themselves to the same class of timing cost model transformations. But in general, there is no reason why a somehow globally optimal transformation should not insert delays elsewhere, possibly at *join* nodes. In such cases (say: C''), an edge-wise comparison of the resulting timing cost model can no longer be expected to yield $C'' \geq C'$ *or* $C' \geq C''$, even though one of these might intuitively be better than the other.

Summary

- Given a set of observable nodes and fixed cost model C, any non-loop timing leaks can be in principle be “transformed out”.

13 Micro-Architectural Dependencies

And by the help of Microscopes, there is nothing so small, as to escape our inquiry; hence there is a new visible World discovered to the understanding.

(Robert Hooke — Micrographia)

Timing based attacks often rely on difference of timing behavior due to *micro-architectural* state, which may in turn depend on private data. Intuitively, given a concrete machine implementing a given ("higher-level", more abstract *macro-architectural*) execution semantics, micro-architectural state is that part of the concrete machine which is "invisible" to the higher level execution semantics. For example, the micro-architectural state of modern CPU includes the state of data- and code *caches* or the state of the CPU's *instruction pipeline* (fetch, decode, execute, etc). The micro-architectural state of two different CPU implementing the same instruction set architecture (i.e.: the same *macro*-architectural semantic) may differ considerably. Consider, for example, AMD and Intel CPUs implementing the x86 instruction set architecture. Still ideally, a x86 program's behavior is the same on both AMD and Intel CPUs, *except for the timing behavior*[1].

In this chapter, I will develop a new method for the dependence analysis of arbitrary micro-architectural state for single-threaded execution models, in form of a new notion $n \to_{\mu\mathrm{d}} m$ of *micro-architectural*

[1] and, of course, difference of behavior due to multi-processing, which I do not cover in this chapter

dependence between nodes n, m. After an introductory example in Section 13.1, I introduce in Section 13.2 the necessary technical framework, and those assumption on the micro-architectural state model necessary for the dependency analysis. In Section 13.6, I define micro-architectural dependence by a reduction to non-termination insensitive control dependence $\rightarrow_{\text{nticd}}$. In Section 13.4, I discuss limitations of my approach, and finally in Section 13.5 I explain how micro-architectural dependence $n \rightarrow_{\mu\text{d}} m$ and timing sensitive control dependence $\rightarrow_{\text{tscd}}$ can be combined to obtain timing sensitive slices that are sensitive to timing channels due to differences in micro-architectural state.

As a running example, I will augment a simple "standard" semantics for CFGs with a simple micro-architectural state, comprised of a simple one-level data cache. I try to make plausible — but give no formal proof — that my method does indeed for work for arbitrary micro-architectural state. For the example cache-semantics, I validated my claims by extensive random testing.

13.1 Introduction

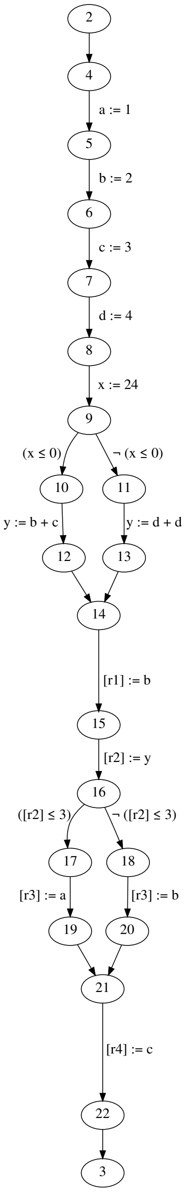
Consider the control flow graph in Figure 13.1a on page 236. Its edges are labeled with assignments and guards that refer to (cachable) variables `a`, `b`, . . ., and uncachable registers `r1`, `r2`, ...

I want to know whether the choice made at node $n = 9$ influences the execution time of the reads of variables `b` and `y` at nodes 14, 15. I assume a simple data cache of size four, with a least recently used eviction strategy. The (micro-architectural) cache-state hence consists of a list $[x_1, x_2, x_3, x_4]$ of variables, with x_1 being the most recently used, and x_4 the next to be evicted. In Figure 13.1b, I show — under an abstraction that considers cache state only — all possible executions of the control flow graph, assuming an empty initial cache. For example, the abstract node $(9, [\mathtt{x}, \mathtt{d}, \mathtt{c}, \mathtt{b}])$ represents all those concrete configurations at control node 9 in which the concrete micro-architectural cache contains cached values for the variables $[\mathtt{x}, \mathtt{d}, \mathtt{c}, \mathtt{b}]$, in this order (with arbitrary concrete macro-architectural state).

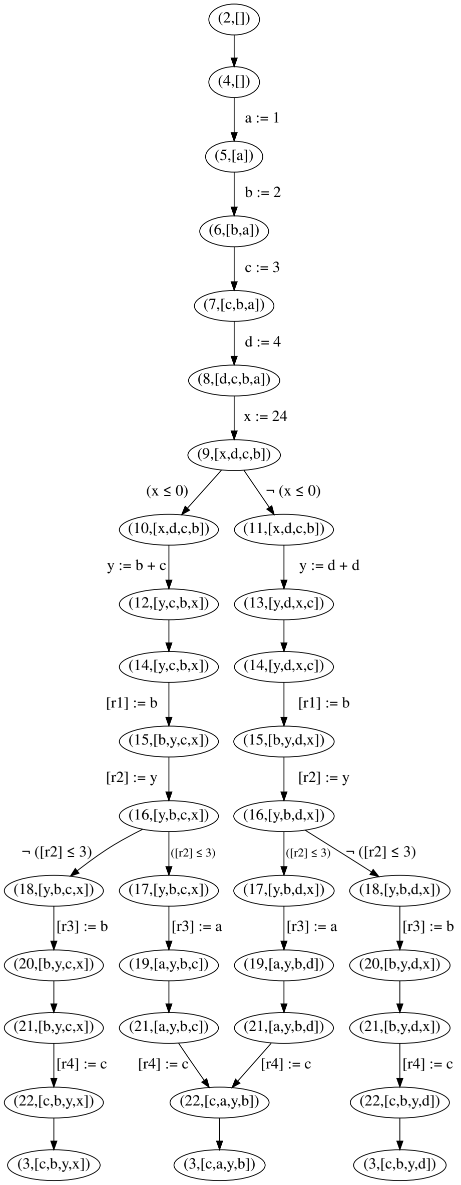
In the example, executions can reach the control node $m = 15$ at cache states represented by either $[\mathtt{b}, \mathtt{y}, \mathtt{c}, \mathtt{x}]$, or by $[\mathtt{b}, \mathtt{y}, \mathtt{d}, \mathtt{x}]$. Which of these (abstract) cache states is reached is determined by the macro-architectural choice made at $n = 9$. But it is easy to see that the execution time of the read of `y` at node $m = 15$ does *not* depend on the choice made at $n = 9$, since in both (classes of) executions that reach node $m = 15$, the cache *does* contain the variable `y`, which is the only cacheable variable accessed by the edge $15 \xrightarrow{\mathtt{r}_2 := \mathtt{y}} 16$ at m.

For the read of variable `b` at node $m = 14$, on the other hand, one class of executions reaches m in $(14, [\mathtt{y}, \mathtt{c}, \mathtt{b}, \mathtt{x}])$ (containing `b`), while another class of executions reaches m in $(14, [\mathtt{y}, \mathtt{d}, \mathtt{x}, \mathtt{c}])$ (*not* containing `b`). Whether the relevant variable `b` is in the cache at $m = 14$ (and hence: the execution time of the read of `b` at $m = 14$) or not depends here on the choice made at $n = 9$.

Now consider the read of `c` at node $m = 21$. Does its cache state depend on the choice made right before at $n' = 16$? There are



(a) Control Flow Graph



(b) Cache Aware Abstract Executions

four (abstract) cache states at $m = 21$. Two contain the variable c: $(21, [\texttt{b}, \texttt{y}, \texttt{c}, \texttt{x}])$ and $(21, [\texttt{a}, \texttt{y}, \texttt{b}, \texttt{c}])$. The other two do *not* contain c: $(21, [\texttt{a}, \texttt{y}, \texttt{b}, \texttt{d}])$ and $(21, [\texttt{b}, \texttt{y}, \texttt{d}, \texttt{x}])$. The cache states containing c are reachable from configurations at control node $n' = 16$. At the same time: cache states *not* containing c are *also* reachable from configurations at control node $n' = 16$. But in fact, whether c is in cache at m does *not* depend on the choice made at n'. To see this, note that node $n' = 16$ can be reached at two different cache states. The first abstract configuration is $(16, [\texttt{y}, \texttt{b}, \texttt{c}, \texttt{x}])$. But whenever $m = 21$ is reached from this abstract configuration, c *is* in the cache (either $(21, [\texttt{b}, \texttt{y}, \texttt{c}, \texttt{x}])$ or $(21, [\texttt{a}, \texttt{y}, \texttt{b}, \texttt{c}])$). The second abstract configuration at which $n' = 16$ can be reached is $(16, [\texttt{y}, \texttt{b}, \texttt{d}, \texttt{x}])$. But whenever $m = 21$ is reached from that configuration, c *is not* in the cache ($(21, [\texttt{a}, \texttt{y}, \texttt{b}, \texttt{d}])$ or $(21, [\texttt{b}, \texttt{y}, \texttt{d}, \texttt{x}])$).

On the other hand, the cache status of c at node $m = 21$ *does* depend on the choice made earlier at $n = 9$. In this example this is necessarily so, since the node $n = 9$ is the only other macro-architectural conditional node in the control flow graph. But this is also directly evident by the structure of the graph in Figure 13.1b.

Remark 13.1.1. With only a small modification of the program, the cache status of c at $m = 21$ could have been *independent* from the choice made earlier at $n = 9$. For example: had there been reads to two additional variables (e.g: $\texttt{e}, \texttt{f}$) right before $m = 21$, then *all* cache states at m would *not* have contained c. This is because these two reads would have evicted c even from $[\texttt{b}, \texttt{y}, \texttt{c}, \texttt{x}]$ (and $[\texttt{a}, \texttt{y}, \texttt{b}, \texttt{c}]$).

In summary, the choice made at $n = 9$ does influence the relevant (micro-architectural) cache state at $m \in \{21, 14\}$. In fact for this micro-architecture, these are the *only* micro-architectural dependencies in this control flow graph G, i.e. I will later say:

$$\rightarrow^{G}_{\mu\mathrm{d}} = \{(9, 21), (9, 14)\}$$

13.2 Control Flow Graphs

In this chapter, I will assume programs given in form of *control flow graphs*, i.e. graphs (N, E) with *labeled* edges $n \xrightarrow{l}_G m \in E$. Labels $l \in L$ may be either assignments or guards. Assignments read and write *macro-architectural* state σ_{M} (e.g.: values of program variables). Guards evaluate macro-architectural state, and then either allow control flow to pass, or not (without changing any macro-architectural state σ_{M}).

Definition 13.2.1. Formally, I assume some set Σ_{M} of possible macro-architectural states σ_{M}, and labels $L = A \cup G$. For each label l I assume a macro-architectural (concrete) interpretation l^{M} of l such that for $a \in A$, a^{M} is a function

$$a^{\mathsf{M}} : \Sigma_{\mathsf{M}} \to \Sigma_{\mathsf{M}}$$

and for $g \in G$, g^{M} is a function

$$g^{\mathsf{M}} : \Sigma_{\mathsf{M}} \to \{\mathbf{false}, \mathbf{true}\}$$

A control flow graph $G = (N, E)$ is said to be (two-branch) deterministic (with regard to macro-architectural state) if

1. whenever $n \xrightarrow{a}_G m$, then this is the only edge leaving n in G
2. whenever $n \xrightarrow{g}_G m$ and $n \xrightarrow{g'} m'$, then either $m = m'$ and $g = g'$, or $m \neq m'$ and the interpretation $(g')^{\mathsf{M}}$ of the label g' is the negation of the interpretation g^{M} of the label g: $(g')^{\mathsf{M}} = \neg \circ g^{\mathsf{M}}$ and.

I assume all control flow graphs to be deterministic in this sense.

The macro-architectural small-steps semantics of control flow graph G is then simply given by

$$\frac{n \xrightarrow{a}_G m \qquad \sigma'_{\mathsf{M}} = a^{\mathsf{M}}(\sigma_{\mathsf{M}})}{(n,\ \sigma_{\mathsf{M}}) \to (m,\ \sigma'_{\mathsf{M}})}\ \text{Ass}$$

$$\frac{n \xrightarrow{g}_G m \qquad g^{\mathsf{M}}\left(\sigma_{\mathsf{M}}\right) = \mathbf{true}}{\left(n,\ \sigma_{\mathsf{M}}\right) \rightarrow \left(m,\ \sigma_{\mathsf{M}}\right)}\ \text{Guard}$$

Later, it will be convenient to assume the macro-architectural to consists of *program variables* $v \in \mathrm{Var}$, and then assume some functions $\mathrm{use}, \mathrm{def} : L \rightarrow 2^{Var}$, with which I can then define the *use* and *def* sets of a node $n \in N$:

$$\begin{aligned} \mathrm{use}\left(n\right) &= \bigcup_{n \xrightarrow{l}_G m} \mathrm{use}\left(l\right) \\ \mathrm{def}\left(n\right) &= \bigcup_{n \xrightarrow{l}_G m} \mathrm{def}\left(l\right) \end{aligned}$$

Definition 13.2.2. I assume a set Σ_μ of micro-architectural states $\sigma_\mu \in \Sigma_\mu$. For each label l, I assume a (concrete) micro-architectural interpretation l^μ of l such that l^μ is a function

$$l^\mu : \Sigma_{\mathsf{M}} \times \Sigma_\mu \rightarrow \Sigma_\mu$$

I also assume for each label l a (concrete) timing cost interpretation $l^{\text{⏱}}$ of l such that $l^{\text{⏱}}$ is a function

$$l^{\text{⏱}} : \Sigma_\mu \rightarrow \mathbb{N}$$

that indicates how much time a transition along an edge labeled with l will take in some micro-architectural state σ_μ.

The full small-steps semantics of a control flow graph G is then given by

$$\frac{n \xrightarrow{a}_G m \qquad \sigma'_{\mathsf{M}} = a^{\mathsf{M}}\left(\sigma_{\mathsf{M}}\right) \qquad \sigma'_\mu = a^\mu\left(\sigma_{\mathsf{M}},\ \sigma_\mu\right) \qquad \Delta t = a^{\text{⏱}}\left(\sigma_\mu\right)}{\left(n,\ \sigma_{\mathsf{M}},\ \sigma_\mu,\ t\right) \rightarrow \left(m,\ \sigma'_{\mathsf{M}},\ \sigma'_\mu,\ t + \Delta t\right)}\ \text{Ass}$$

$$\frac{n \xrightarrow{g}_G m \qquad g^{\mathsf{M}}\left(\sigma_{\mathsf{M}}\right) = \textbf{true} \qquad \sigma'_{\mu} = g^{\mu}\left(\sigma_{\mathsf{M}},\, \sigma_{\mu}\right) \qquad \Delta t = g^{\text{⏱}}\left(\sigma_{\mu}\right)}{\left(n,\, \sigma_{\mathsf{M}},\, \sigma_{\mu},\, t\right) \to \left(m,\, \sigma_{\mathsf{M}},\, \sigma'_{\mu},\, t + \Delta t\right)}\,\text{Guard}$$

Note that this semantics is deterministic if G is (in the macro-architectural sense), since only the macro-architectural state decides which CFG edge is traversed.

Remark 13.2.1. The micro-architectural state will often (to some extend) *track* the macro-architectural state. For example, the micro-architectural state of a data cache will include copies the value of those (macro-architectural) variables that are currently cached. Hence the concrete micro-architectural state space for a given program (i.e.: for a control flow graph with designated initial node) can become arbitrarily large.

But since my approach will require me to compute (a representation of) all micro-architectural state of a given program, I require an *abstract* representation $\sigma^{\#}_{\mu}$ of micro-architectural states (with a corresponding abstract interpretation of labels l), which still allows me to assign a definite timing cost to each label l.

Definition 13.2.3 (Abstract Micro-Architectural Semantics)**.** Let α be a (abstraction) function

$$\alpha : \Sigma_{\mu} \to \Sigma^{\#}_{\mu}$$

from (concrete) micro-architectural states σ_{μ} to some abstraction $\alpha\left(\sigma_{\mu}\right)$ in some set $\Sigma^{\#}_{\mu}$ of abstract micro-architectural states.

I write

$$\gamma(\sigma^{\#}_{\mu}) = \{\, \sigma_{\mu} \mid \alpha\left(\sigma_{\mu}\right) = \sigma^{\#}_{\mu} \,\}$$

Recall that in the full concrete small-step semantic (Definition 13.2.2), the choice which control flow graph edge to traverse next depends only on the macro-architectural state. Also, given such a choice (e.g.: $n \xrightarrow{l}_G m$), the concrete micro-architectural choice is deterministic.

The abstract micro-architectural small-steps semantics of a control flow graph G is then just

$$\frac{n \xrightarrow{l}_G m \qquad \alpha\left(\sigma_\mu\right) = \sigma_\mu^\# \qquad \sigma'_\mu = l^\mu\left(\sigma_{\mathsf{M}},\, \sigma_\mu\right) \qquad \alpha\left(\sigma'_\mu\right) = \sigma'^\#_\mu}{\left(n,\, \sigma_\mu^\#\right) \xrightarrow{l} \left(m,\, \sigma'^\#_\mu\right)}\ \text{Label}$$

Even for a fixed choice $n \xrightarrow{l}_G m$, the abstract micro-architectural small-steps semantics may be indeterministic. For example, $\sigma'^\#_\mu$ may differ for different σ_{M}. For an example, see Section 13.6.

Remark 13.2.2. In the data-cache example, a concrete cache σ_μ (mapping *some* program variables to values, and ordering these variables by recency of use) can be abstractly represented by the list $\alpha_{\text{cache}}\left(\sigma_\mu\right)$ of variables currently in the cache. Using a different abstraction, a concrete cache σ_μ can also be abstractly represented by the set $\alpha_{\text{incache}}^{\text{use}(m)}\left(\sigma_\mu\right)$ of variables used at some node m that are in cache at state σ_μ.

In Figure 13.1b, I showed the abstract micro-architectural small-steps semantics for the control flow graph in Figure 13.1a, for all abstract configurations reachable from the abstract configuration $(2,\ [])$. I used the abstraction α_{cache}.

Definition 13.2.4. A function

$$\alpha^{\text{⏱}} : \Sigma_\mu \to \Sigma_\mu^{\text{⏱}}$$

from (concrete) micro-architectural states σ_μ to some abstraction $\alpha^{\text{⏱}}\left(\sigma_\mu\right)$ in some set $\Sigma_\mu^{\text{⏱}}$ of abstract micro-architectural states is said to respect timing cost for label l if

for all σ_μ, σ'_μ with $\alpha^{\text{⏱}}\left(\sigma_\mu\right) = \alpha^{\text{⏱}}\left(\sigma'_\mu\right)$ I have:

$$l^{\text{⏱}}\left(\sigma_\mu\right) = l^{\text{⏱}}\left(\sigma'_\mu\right)$$

Remark 13.2.3. The abstraction α_{cache} respects timing cost for all labels, while the coarser abstraction $\alpha_{\text{incache}}^{\text{use}(m)}$ respects all labels at control flow graph edges leaving m. This is because I assume that the time it takes to access a variable x does only depend on whether x is currently cached or not, but not on the concrete *value* of x (or its position in the cache).

I require one property of abstractions α. Intuitively, the abstraction must *not* conflate two different micro-architectural states σ_μ^1 and σ_μ^2 if these may lead to different execution times from configurations consisting of these two micro-architectural states. This is necessary because if whenever such micro-architectural states are conflated by α i.e.:

$$\alpha\left(\sigma_\mu^1\right) = \alpha\left(\sigma_\mu^2\right) = \sigma_\mu^\#$$

then these may introduce *join nodes* $(m, \sigma_\mu^\#)$ in the graph obtained from the abstract micro-architectural semantics. But my notion of micro-architectural dependencies will then judge this *join node*[2] to be independent from choices made before at conditional nodes $(n, \ldots)$ whose successors have all joined in $(m, \sigma_\mu^\#)$, which is unsound if executions continuing from there may have different timing behavior, possibly due the actual choice made at n.

Definition 13.2.5. An abstraction function α is said to respect timing cost *for all possible execution* if for any two micro-architectural states σ_μ^1 and σ_μ^2 that are conflated by α, any two executions starting at full concrete configurations which differ only in these two micro-architectural states are indiscernible "up to micro-architectural state".

Formally: For all micro-architectural states σ_μ^1 and σ_μ^2 such that

$$\alpha\left(\sigma_\mu^1\right) = \alpha\left(\sigma_\mu^2\right)$$

[2] and following nodes

and all nodes n, macro-architectural states σ_{M} and points in time $t \in \mathbb{N}$, whenever

$$\begin{aligned} &\left(n,\ \sigma_{\mathsf{M}},\ \sigma_{\mu}^{1},\ t\right) \to \ldots \to \left(m,\ \sigma_{\mathsf{M}}^{1},\ \sigma_{\mu}^{1'},\ t_1\right) \\ \text{and}\quad &\left(n,\ \sigma_{\mathsf{M}},\ \sigma_{\mu}^{2},\ t\right) \to \ldots \to \left(m,\ \sigma_{\mathsf{M}}^{2},\ \sigma_{\mu}^{2'},\ t_2\right) \end{aligned}$$

with no other occurrences of m in the transition sequences, then $\sigma_{\mathsf{M}}^{1} = \sigma_{\mathsf{M}}^{2}$ and $t_1 = t_2$, as well as $\alpha\left(\sigma_{\mu}^{1'}\right) = \alpha\left(\sigma_{\mu}^{2'}\right)$.

Remark 13.2.4. In Definition 13.2.5, $\sigma_{\mathsf{M}}^{1} = \sigma_{\mathsf{M}}^{2}$ must already hold simply due to the fact that both the choice of successor-nodes and the macro-architectural successor state are independent from micro-architectural state (and the fact that the control flow graph is assumed to be deterministic). $t_1 = t_2$ must already hold if α respects timing cost for all labels.

13.3 Micro-Architectural Dependencies

My ultimate goal is to define, given a control flow graph $G = (N, E)$, a binary relation $\rightarrow^{G}_{\mu\mathrm{d}}$ on nodes $n, m \in N$ such that

$$n \rightarrow^{G}_{\mu\mathrm{d}} m$$

if the (macro-architectural) choice made at n influences the timing behavior at node m, *if* m is executed (after n)[3].

Since the timing behavior at node m depends (by Definition 13.2.2) only on the *micro*-architectural state, I need to determine if the choice made at n influences micro-architectural state at node m. In other words: for those labels l at control flow graph edges $m \xrightarrow{l} m'$ leaving m, I am interested in whether the choice made at n influences the timing behavior $l^{⏱}$ of micro-architectural states σ_μ at m.

The general idea that I develop in this chapter is to define $\rightarrow^{G}_{\mu\mathrm{d}}$ in terms of nontermination insensitive control dependence $\rightarrow_{\mathrm{nticd}}$, but not on the control flow graph G, but on a graph derived from the abstract micro-architectural small steps semantics

$$\left(n,\ \sigma^{\#}_{\mu}\right) \xrightarrow{l} \left(n',\ \sigma'^{\#}_{\mu}\right)$$

as defined in Definition 13.2.3, and previously exemplified in Figure 13.1b on page 236.

The informal slogan

$$\rightarrow^{G}_{\mu\mathrm{d}} \quad \approx \quad |N| \times \rightarrow^{G_{\#}}_{\mathrm{nticd}}$$

[3] The question whether m is executed at all after n, and whether this depends on the choice made at n, is already answered by control dependence and related notions.

expresses that $\rightarrow^{G}_{\mu\mathrm{d}}$ can be obtained from $|N|$ applications of nontermination-insensitive control dependence $\rightarrow_{\mathrm{nticd}}$ on graphs $G_{\#}$ derived from G (and $m \in N$): One application for each node $m \in N$.

Definition 13.3.1. Fix some deterministic control flow graph $G = (N, E)$ and some (initial) node $n^0 \in N$ in G as well as some (initial) concrete micro-architectural state $\sigma^0_\mu \in \Sigma_\mu$.

Let $N^{\#}$ be the set of configurations $\left(n, \sigma^{\#}_\mu\right)$ reachable from $\left(n^0, \alpha\left(\sigma^0_\mu\right)\right)$ in the abstract micro-architectural small steps semantics (Definition 13.2.3).

Then the graph $G_\alpha = (N_\alpha, E_\alpha)$ consists of the nodes

$$N_\alpha \;=\; \{ \left(n, \gamma(\sigma^{\#}_\mu)\right) \mid \left(n, \sigma^{\#}_\mu\right) \in N^{\#} \} \;\subseteq\; N \;\times\; 2^{\Sigma_\mu}$$

and L-labeled edges

$$E_\alpha \;=\; \{ \left((n, \gamma(\sigma^{\#}_\mu)),\, l,\, (m, \gamma(\sigma'^{\#}_\mu))\right) \mid (n, \sigma^{\#}_\mu) \in N^{\#},\, (n, \sigma^{\#}_\mu) \xrightarrow{l} (m, \sigma'^{\#}_\mu) \}$$

From now on, I assume G_α to be finite. Also from now on, I fix a node $m \in N$ in the control flow graph G, and let $\alpha^{⏱}$ be an abstraction coarser than α that respects timing cost for all labels l at edges leaving $m \in G$.

I write $M^{\#} = \{(m, \sigma^{\#}_\mu) \in N^{\#}\}$ for the set of abstract configurations at the fixed node m, and

$$M = \{ (m, \gamma(\sigma^{\#}_\mu)) \mid (m, \sigma^{\#}_\mu) \in M^{\#} \}$$

for the set of corresponding nodes in G_α.

Then I define

$$G_{\alpha,m} = \left(G_\alpha^{\rightarrow^* M}\right)_{M \not\rightarrow}$$

as the graph obtained from G_α by removing all nodes that cannot reach M, and all edges leaving M. I also define the graph

$$G_{\alpha,m,\alpha^{⏱}}$$

as the graph obtained from $G_{\alpha,m}$ by merging for each $\sigma_\mu^{⏱} \in \Sigma_\mu^{⏱}$ those nodes (m, Σ) at m such that for all $\sigma_\mu \in \Sigma$, σ_μ is represented in $\Sigma_\mu^{⏱}$ by $\sigma_\mu^{⏱}$ (i.e.: $\alpha^{⏱}(\sigma_\mu) = \sigma_\mu^{⏱}$).

In Figure 13.1b on page 236, I previously showed the graph G_α for start node $n^0 = 2, \alpha = \alpha_{\text{cache}}$, and the initially empty cache σ_μ^0. Now in Figure 13.2a on page 247, I show the graph $G_{\alpha,m,\alpha^{⏱}}$ for $m = 14$ and $\alpha^{⏱} = \alpha_{\text{incache}}^{\text{use}(m)}$, and in Figure 13.2b I do the same for $m = 21$.

For example in G_α, I had two nodes $(14, \gamma([\texttt{y}, \texttt{c}, \texttt{b}, \texttt{x}]))$ and $(14, \gamma([\texttt{y}, \texttt{d}, \texttt{c}, \texttt{x}]))$. On the other hand in $G_{\alpha,14,\alpha^{⏱}}$, the abstraction

$$\alpha^{⏱} = \alpha_{\text{incache}}^{\text{use}(14)} = \alpha_{\text{incache}}^{\{\texttt{b}\}}$$

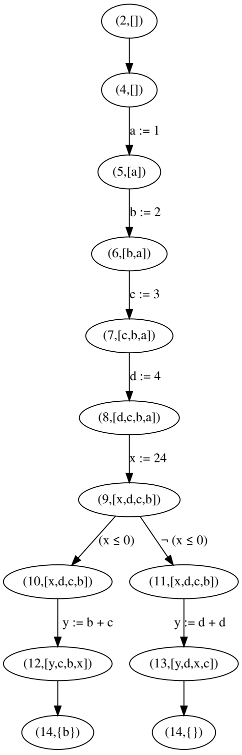
considers only whether the variable `b` is in the cache. Hence I there have two nodes $(14, \{\texttt{b}\})$ and $(14, \{\})$.

Similarly in G_α, I had four nodes $(21, \gamma([\texttt{b}, \texttt{y}, \texttt{c}, \texttt{x}]))$, $(21, \gamma([\texttt{a}, \texttt{y}, \texttt{b}, \texttt{c}]))$, $(21, \gamma([\texttt{a}, \texttt{y}, \texttt{b}, \texttt{d}]))$ and $(21, \gamma([\texttt{b}, \texttt{y}, \texttt{d}, \texttt{x}]))$. On the other hand in $G_{\alpha,21,\alpha^{⏱}}$, the abstraction

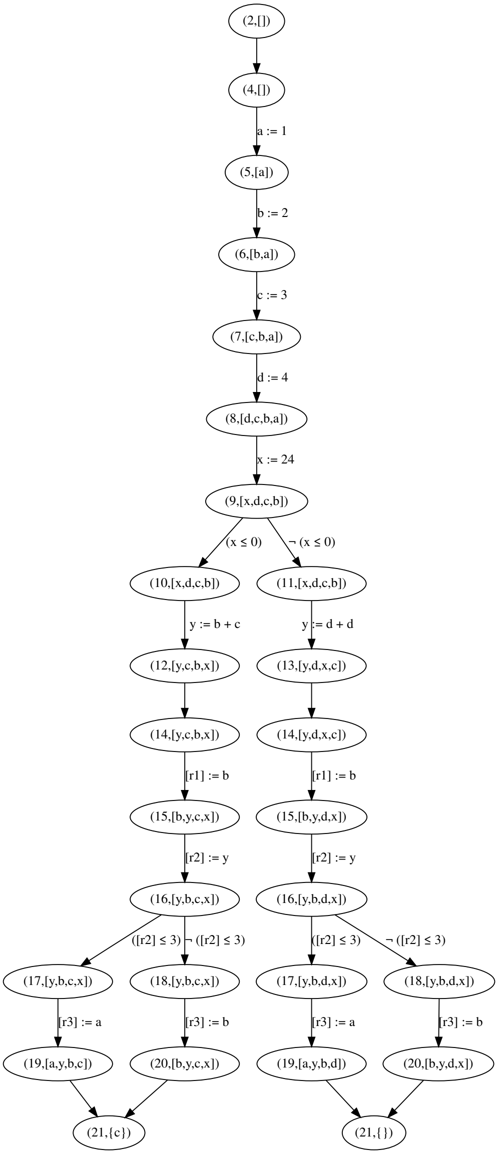
$$\alpha^{⏱} = \alpha_{\text{incache}}^{\text{use}(21)} = \alpha_{\text{incache}}^{\{\texttt{c}\}}$$

considers only whether the variable `c` is in the cache. Hence I there have two nodes $(21, \{\texttt{c}\})$ and $(21, \{\})$.

Evidently, those nodes (m, Σ) for which $m = 14$, are control dependent on the node $(9, [\texttt{x}, \texttt{d}, \texttt{c}, \texttt{b}]))$ in $G_{\alpha,14,\alpha^{⏱}}$, but no other node. Similarly, those nodes (m, Σ) for which $m = 21$, are control dependent on the



(a) The Graph $G_{\alpha,14,\alpha^{⏱}}$



(b) The Graph $G_{\alpha,21,\alpha^{⏱}}$

same node $(9, [\mathtt{x}, \mathtt{d}, \mathtt{c}, \mathtt{b}]))$ in $G_{\alpha,21,\alpha^{⏱}}$, but no other node. This is just as expected, since I had predicted:

$$\rightarrow^{G}_{\mu\mathrm{d}} = \{(9,21), (9,14)\}$$

Let me now make some general observations for the nontermination insensitive postdominance relation in $\sqsupseteq^{G_{\alpha,m,\alpha^{⏱}}}_{\mathrm{SINK}}$ in the graph $G_{\alpha,m,\alpha^{⏱}}$ for a fixed node m.

- The transitive reduction $<^{G_{\alpha,m,\alpha^{⏱}}}_{\mathrm{SINK}}$ is a forest. Its roots include the nodes (m, Σ), i.e: those nodes (n', Σ) with $n' = m$. All roots are singular nodes (i.e.: there are only trivial control-sinks).
- The conditional nodes (n, Σ) in $G_{\alpha,m,\alpha^{⏱}}$ correspond to either conditional nodes n in G, or nodes at which the concrete macro-architectural state σ_{M} may influence the abstract micro-architectural successor state $\sigma'^{\#}_{\mu}$ (see Section 13.6 for an example).
- Because all outgoing edges of M were deleted in $G_{\alpha,m}$, for each node (n, Σ_n), at most one node (m, Σ) postdominates (n, Σ_n):

$$\left| \{\, (m, \Sigma) \mid (m, \Sigma) \sqsupseteq^{G_{\alpha,m,\alpha^{⏱}}}_{\mathrm{SINK}} (n, \Sigma_n) \,\} \right| \leq 1$$

- If some (m, Σ) postdominates some (n, Σ_n), then all concrete executions first reaching a state represented by (n, Σ_n), and then next reaching a state at m, have the same timing behavior at m. This is because all $\sigma_\mu \in \Sigma$ have the same timing behavior (see Definition 13.2.4).
- Micro-architectural dependency are *not* meant to include "normal" control dependencies. Hence it is intentional that even if some (m, Σ) postdominates some (n, Σ_n) in $G_{\alpha,m,\alpha^{⏱}}$, choices at n may still decide whether a concrete execution reaches m at all (i.e.: m may still be *control dependent* on n in G). This is be-

cause $G_{\alpha,m,\alpha^{⏱}}$ only consists of nodes that "eventually" (in the nontermination insensitively sense) must reach m (at *some* micro-architectural state σ_μ). This was achieved by taking $G_{\alpha}^{\rightarrow^* M}$.

- If *no* node (m, Σ) postdominates a given node (n, Σ_n), but some other node $(n', \Sigma_{n'})$ *does* postdominate (n, Σ_n), i.e., if:

  $$(n', \Sigma_{n'}) \sqsupseteq_{\text{SINK}}^{G_{\alpha,m,\alpha^{⏱}}} (n, \Sigma_n)$$

  (with $n' \neq m$, $n' \neq n$), then any decision made at n cannot possibly influence the micro-architectural state at m, since all concrete executions from states represented by (n, Σ_n) have then joined again at a state represented by $(n', \Sigma_{n'})$ *before* reaching m.

But this last remark just describes the converse of

$$\rightarrow_{\text{ntind}}^{G_{\alpha,m,\alpha^{⏱}}} = \left(\rightarrow_{\text{nticd}}^{G_{\alpha,m,\alpha^{⏱}}} \right)^*$$

(see Definition 7.1.1 on page 122, and Observation 7.1.1). Hence I *could* just define[4]

$$n \rightarrow_{\mu\text{d}}^{G} m \quad \iff \quad \exists (n, \Sigma_n), (m, \Sigma) \in N_{\alpha,m,\alpha^{⏱}}.\ (n, \Sigma_n) \rightarrow_{\text{ntind}}^{G_{\alpha,m,\alpha^{⏱}}} (m, \Sigma)$$

and obtain a sound notion of micro-architectural dependencies. But it turns out that this notion would be too coarse, because it distinguishes too many abstract states (n, Σ_n).

To see this, consider Figure 13.3b on page 251, which shows the graph $G_{\alpha,26,\alpha^{⏱}}$ for the control flow graph G in Figure 13.3a. The question is if $m = 26$ is micro-architecturally dependent on $n = 10$, i.e. if

$$10 \rightarrow_{\mu\text{d}}^{G} 26$$

[4] with a misuse of notation: the names n, m are not meant to be bound by $\exists$, but only the names Σ, Σ_n

If I were to answer this directly by consulting $G_{\alpha,26,\alpha^{⏱}}$, I would have to affirm this, since I have

$$(10, \text{␣}) \rightarrow^{G_{\alpha,26,\alpha^{⏱}}}_{\text{nticd}} (18, \text{␣}) \rightarrow^{G_{\alpha,26,\alpha^{⏱}}}_{\text{nticd}} (26, \text{␣}) \text{ hence } (10, \text{␣}) \rightarrow^{G_{\alpha,26,\alpha^{⏱}}}_{\text{ntind}} (26, \text{␣})$$

But in reality, node $n = 10$ can *not* influence whether $\texttt{x}$ is in cache at node $m = 26$. This is because any concrete execution at a concrete configuration represented by $(18, [\texttt{a}, \texttt{z}, \texttt{d}, \texttt{c}])$ will make the same choices as a concrete execution at a concrete configuration represented by $(18, [\texttt{a}, \texttt{z}, \texttt{c}, \texttt{d}])$[5]: if $(18, [\texttt{a}, \texttt{z}, \texttt{d}, \texttt{c}])$ proceeds along the edge labeled $l = 0 \leq \texttt{r}_0 \cdot \texttt{r}_1$ (to $(19, [\texttt{a}, \texttt{z}, \texttt{d}, \texttt{c}])$), then $(18, [\texttt{a}, \texttt{z}, \texttt{c}, \texttt{d}])$ will *also* proceed along an edge labeled l (to $(19, [\texttt{a}, \texttt{z}, \texttt{c}, \texttt{d}])$). But it is now easy to see that $(19, [\texttt{a}, \texttt{z}, \texttt{c}, \texttt{d}]))$ and $(19, [\texttt{a}, \texttt{z}, \texttt{d}, \texttt{c}]))$ behave equivalently with regard to the relevant cache state at $m = 26$, since from both these abstract configuration, execution must reach a configuration at which the relevant variable $\texttt{c}$ is *not* in cache, i.e.:

$$\begin{aligned} &(26, []) \sqsupseteq^{G_{\alpha,26,\alpha^{⏱}}}_{\text{SINK}} (19, [\texttt{a}, \texttt{z}, \texttt{c}, \texttt{d}]) \\ \text{and} \quad &(26, []) \sqsupseteq^{G_{\alpha,26,\alpha^{⏱}}}_{\text{SINK}} (19, [\texttt{a}, \texttt{z}, \texttt{d}, \texttt{c}]) \end{aligned}$$

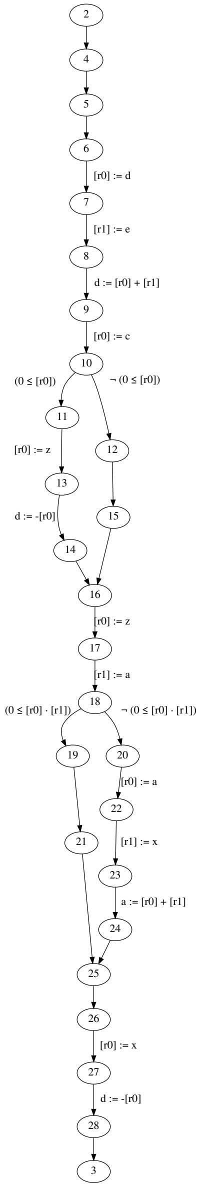
Similarly, for the other edge $l' = \neg\ l$ at $(18, [\texttt{a}, \texttt{z}, \texttt{c}, \texttt{d}])$ and $(18, [\texttt{a}, \texttt{z}, \texttt{d}, \texttt{c}])$, I have for their respective successors at 20:

$$\begin{aligned} &(26, [\texttt{x}]) \sqsupseteq^{G_{\alpha,26,\alpha^{⏱}}}_{\text{SINK}} (20, [\texttt{a}, \texttt{z}, \texttt{c}, \texttt{d}]) \\ \text{and} \quad &(26, [\texttt{x}]) \sqsupseteq^{G_{\alpha,26,\alpha^{⏱}}}_{\text{SINK}} (20, [\texttt{a}, \texttt{z}, \texttt{d}, \texttt{c}]) \end{aligned}$$

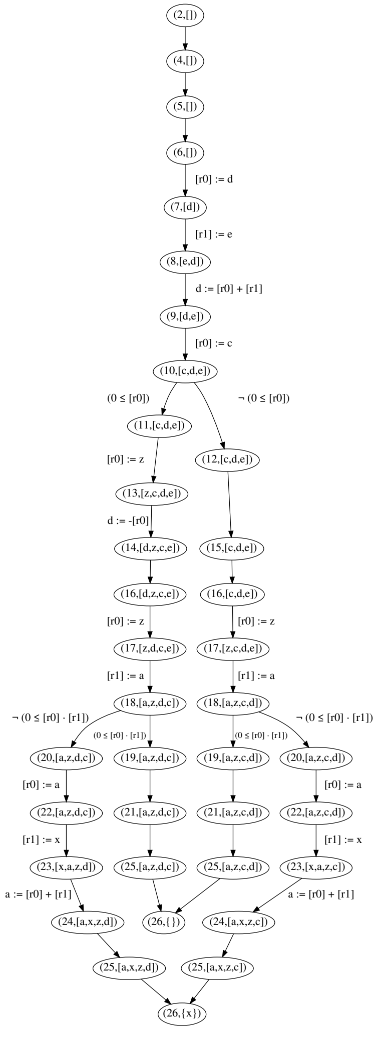
In other words, both successor configurations must reach node $m = 26$ at a configuration in which $\texttt{x}$ is in the cache.

In summary: while the macro-architectural node $n = 10$ does indeed influence in whether execution reaches node 18 at $(18, [\texttt{a}, \texttt{z}, \texttt{d}, \texttt{c}])$ or at $(18, [\texttt{a}, \texttt{z}, \texttt{c}, \texttt{d}])$, this choice at $n = 10$ is irrelevant for the (timing

[5] note the different order of $\texttt{c}, \texttt{d}$



(a) A Control Flow Graph G



(b) The Graph $G_{\alpha,26,\alpha^{⏱}}$

relevant) micro-architectural state at node $m = 26$. In this sense (with regard to node $m = 26$), $(18, [\mathtt{a}, \mathtt{z}, \mathtt{d}, \mathtt{c}])$ and $(18, [\mathtt{a}, \mathtt{z}, \mathtt{c}, \mathtt{d}])$ are *equivalent*: if in a concrete configuration abstracted by $(18, [\mathtt{a}, \mathtt{z}, \mathtt{d}, \mathtt{c}])$ and with macro-architectural state σ_{M}, the concrete execution proceeds towards a configuration at $m = 26$ with $\mathtt{x}$ in cache, then any concrete execution from a concrete configuration abstracted by $(18, [\mathtt{a}, \mathtt{z}, \mathtt{c}, \mathtt{d}])$ with the same macro-architectural state σ_{M} will *also* proceed towards a configuration at $m = 26$ with $\mathtt{x}$ in cache.

What I need then is a general notion $\equiv$ of equivalence between nodes (n, Σ_n), $(n, \Sigma_{n'})$ in $G_{\alpha,m,\alpha^{⏱}}$.

Intuitively, one might want to make use specifically of the fact that in this example, the abstract cache states $[\mathtt{a}, \mathtt{z}, \mathtt{d}, \mathtt{c}]$ and $[\mathtt{a}, \mathtt{z}, \mathtt{c}, \mathtt{d}]$ are in some sense equivalent at configurations with control flow graph node $n' = 18$. This *can* be done, but one has to be very careful to find the correct equivalence here, since whether $[\mathtt{a}, \mathtt{z}, \mathtt{c}, \mathtt{d}]$ and $[\mathtt{a}, \mathtt{z}, \mathtt{c}, \mathtt{d}]$ "behave equivalently" depends very much their position in, and the structure of the control flow graph. Specifically: it depends on the set of variables accessible between their position and nodes at m. But these considerations are completely unnecessary, since I can easily read off the only kind of equivalence that matters to me *directly* from the structure of $G_{\alpha,m,\alpha^{⏱}}$. All that matters is whether "up to decisions made at conditional nodes" (which are always *independent* of micro-architectural state), two nodes must reach an equivalent micro-architectural state at m. But this information is readily available in the postdominance relation $\sqsupseteq_{\mathrm{SINK}}^{G_{\alpha,m,\alpha^{⏱}}}$. I do *not* need to further inspect the sets Σ at nodes $(x, \Sigma) \in N_{\alpha,m,\alpha^{⏱}}$.

Definition 13.3.2. For a fixed control flow graph node $m \in N$, write $G_\# = (N_\#, E_\#)$ for $G_{\alpha,m,\alpha^{⏱}}$. Let $M = \{\, (m, \Sigma) \mid (m, \Sigma) \in N_\# \,\}$ be the set of nodes in $G_\#$ that represent a configuration at control flow graph node m. Also note that whenever

$$(n, \Sigma_n) \xrightarrow{l}_{G_\#} (x, \Sigma_x)$$

then also

$$(n, \Sigma'_n) \xrightarrow{l}_{G_\#} (x, \Sigma'_x)$$

for some Σ'_x. I.e.: for any $G_\#$-node (n, Σ'_n) with the same G-node n, but (possibly) different set Σ'_n of abstract micro-architectural state, any edge labeled l to some node with the G-node x is matched by some edge also labeled l towards a node with the same G-node x.

Write E both for the following rule system, and the corresponding monotone functional.

$$\frac{}{(n, \Sigma_n) \equiv (n, \Sigma_n)}\ \text{Base}$$

$$\frac{(m, \Sigma) \sqsupseteq^{G_\#}_{\text{SINK}} (n, \Sigma_n) \qquad (m, \Sigma) \sqsupseteq^{G_\#}_{\text{SINK}} (n, \Sigma'_n)}{(n, \Sigma_n) \equiv (n, \Sigma'_n)}\ \sqsupseteq_{\text{SINK}}$$

$$\frac{\begin{array}{c} n \neq m \\ \forall l.\ \Big((n, \Sigma_n) \xrightarrow{l}_{G_\#} (x, \Sigma_x) \wedge (n, \Sigma'_n) \xrightarrow{l}_{G_\#} (x, \Sigma'_x) \ \Rightarrow\ (x, \Sigma_x) \equiv (x, \Sigma'_x) \Big) \end{array}}{(n, \Sigma_n) \equiv (n, \Sigma'_n)}\ \text{Suc}$$

Then define

$$G^{\equiv}_{\alpha,m,\alpha^{⏱}}$$

to be the graph obtained from $G_\# = G_{\alpha,m,\alpha^{⏱}}$ by merging all nodes which *cannot be distinguished* by the rule system E, i.e.: by merging all nodes equivalent w.r.t $\equiv$, where

$$\equiv \quad = \quad \nu\mathsf{E}$$

is the greatest fixed point of E. Here, the greatest fixed point is to be taken with respect to the subset order $\subseteq$, and

$$\top \quad = \quad \{\, (\, (n, \Sigma_n),\ (n, \Sigma'_n)) \mid (n, \Sigma_n) \in N_\#,\ (n, \Sigma'_n) \in N_\# \,\}$$

Rule Base states that every $G_\#$-node (n, Σ_n) is equivalent to itself. Rule $\exists_{\text{SINK}}$ states that two $G_\#$-nodes (n, Σ_n) and (n, Σ'_n) with a common control flow graph node n are equivalent if they are postdominated by the same $G_\#$-node (m, Σ_m), with m being the fixed G-node for which $G_\# = G_{\alpha,m,\alpha^{\text{⏱}}}$. The last rule Suc states that two $G_\#$-nodes (n, Σ_n) and (n, Σ'_n) with a common control flow graph node n are equivalent if for each label l, all l-successors with a common control flow graph node x are equivalent.

Now finally, I can define micro-architectural dependencies.

Definition 13.3.3. Fix some deterministic control flow graph $G = (N, E)$ and some node $n^0 \in N$ in G as well as some (concrete) micro-architectural state $\sigma^0_\mu \in \Sigma_\mu$.

Let $m \in N$ be some node in G, and let $G^{\equiv}_{\alpha,m,\alpha^{\text{⏱}}}$ be as in Definition 13.3.2. Then I define for any node $n \in N$: [6]

$$n \rightarrow^{G}_{\mu\text{d}} m \quad \iff \quad \exists (n, \Sigma_n), (m, \Sigma) \in N^{\equiv}_{\alpha,m,\alpha^{\text{⏱}}}.\ (n, \Sigma_n) \rightarrow^{G^{\equiv}_{\alpha,m,\alpha^{\text{⏱}}}}_{\text{ntind}} (m, \Sigma)$$

For each $m \in N$, computation of $\cdot \rightarrow^{G}_{\mu\text{d}} m$ is efficient. Specifically, since the definition is of the form

$$\left(\rightarrow^{(G^{\equiv}_{\#})_{M\not\to}}_{\text{nticd}} \right)^{*} (M)$$

Lemma 7.1.1 from page 135 applies, and I need to compute neither $\rightarrow_{\text{ntind}}$ nor $\rightarrow_{\text{nticd}}$.

Also in my experiments, computation of $G^{\equiv}_{\alpha,m,\alpha^{\text{⏱}}}$ from $G_{\alpha,m,\alpha^{\text{⏱}}}$ using a chaotic fixed point iteration in topological order is efficient, even when using a naive explicit representation of the equivalence classes in $G_{\alpha,m,\alpha^{\text{⏱}}} / \equiv$.

[6] where again, only Σ_n, Σ, but not m, n are meant to be bound by $\exists$

Of course in practice, computation of $\cdot \rightarrow^{G}_{\mu\mathrm{d}} m$ is only feasible if the computation of G_α is, from which then $G_{\alpha,m,\alpha^{⏱}}$ is easily obtained for each $m \in M$.

13.4 Limitations of Micro-Architectural Dependencies

The requirement of α respecting timing for all possible executions (Definition 13.2.5) is quite strong. Obeying it will lead to huge graphs G_α in some applications. In the micro-architectural cache architecture, for some programs the computation is feasible only for relatively small cache sizes.

By Definition 13.2.2, I assume that the execution time along an edge labeled l depends on the micro-architectural state only. In many CPU architectures, however, there exist instructions whose execution time will depend on the actual *value* of some operand, which is intuitively part of the macro-architectural. For example, the execution of trigonometric operations $(\sin, \cos, \ldots)$ and integer division `div` usually depends on the actual value of the operand. To what extend is this a limitation of my approach? I give three answers:

1. In principle, this is not a limitation at all, since the concrete micro-architectural state σ_μ can always be modeled to include the whole macro-architectural state.
2. However in practice — respecting timing for all possible executions — this *does* appear to be a severe limitation, since I will then be forced to track the actual values of variables in the *abstract* micro-architectural state $\sigma^{\#}_{\mu}$.
3. Then again in practice, I can just ignore these particular kind of execution time dependency *in the micro-architectural dependency graph* $\to^{G}_{\mu\mathrm{d}}$, and defer the obligation to account for these kind of dependencies to the *timing sensitive control dependence* $\to_{\mathrm{tscd}}$, and data dependence $\to_{\mathrm{data}}$. I can do this by a trivial modification of the underlying control flow graph G. Consider an edge $n \xrightarrow{\texttt{r}_0\ :=\ \texttt{x div y}}_G m$, with m being the only G-successor of n. In-

troducing two fresh nodes m_l, m_r, I can just replace this edge by four new edges

$$\begin{aligned} & n \xrightarrow{\texttt{r}_0 \texttt{ := x div y}}_{G'} m_l \rightarrow_{G'} m \\ \text{and} \quad & n \xrightarrow{\texttt{r}_0 \texttt{ := x div y}}_{G'} m_r \rightarrow_{G'} m \end{aligned}$$

obtaining a new control flow graph G', and assigning two different costs to these edges in the underlying timing cost model C' for G'. Then m will be timing sensitively control dependent on n in G' under C'

$$n \rightarrow_{\text{tscd}}^{G'[C']} m$$

and if n is data-dependent on some other node n', then n' will be included in the backward slice of m, as required.

In the next section, I explain how a similar construction can be used to integrate the micro-architectural dependencies $n \rightarrow_{\mu\text{d}}^{G} m$ into a full timing sensitive slice, in general.

13.5 Timing Dependence for Micro-Architectural Dependencies

In a labeled control flow graph $G = (N, E)$, a micro-architectural dependency $n \rightarrow^{G}_{\mu\text{d}} m$ indicates that the time some step at m takes to execute may depend on a (macro-architectural) choice made at n. On the other hand, a *timing* sensitive control dependence $n \rightarrow^{G}_{\text{tscd}} m$ indicates: when *and* if some node m is executed may depend on a choice made at n. But that a step at node m may take different amount of time *at all* is usually not encoded in the structure of the control flow graph, and hence invisible to timing sensitive control dependence $n \rightarrow^{G}_{\text{tscd}} m$. Recall the control flow graph G from Figure 13.1a, repeated in Figure 13.4a on page 259.

I previously found that

$$\rightarrow^{G}_{\mu\text{d}} = \{(9, 21), (9, 14)\}$$

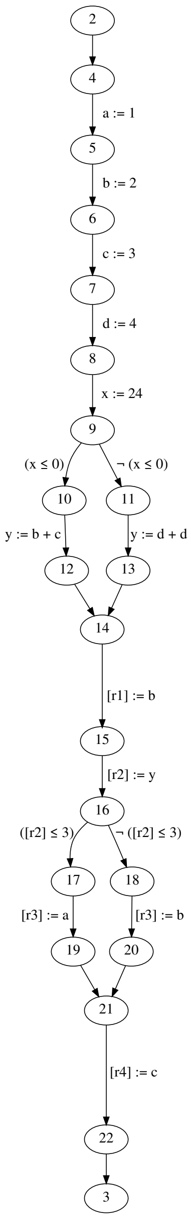
But if I wanted to find out whether the execution of the whole program depended on the choice made at $n = 9$, and did this by naively applying timing sensitive control dependence[7] to G, I might query the timing-sensitive backward slice of the end node $M = \{3\}$, finding that

$$\left(\rightarrow^{G}_{\text{tscd}} \cup \rightarrow^{G}_{\text{data}}\right)^{*}(M) = \emptyset$$

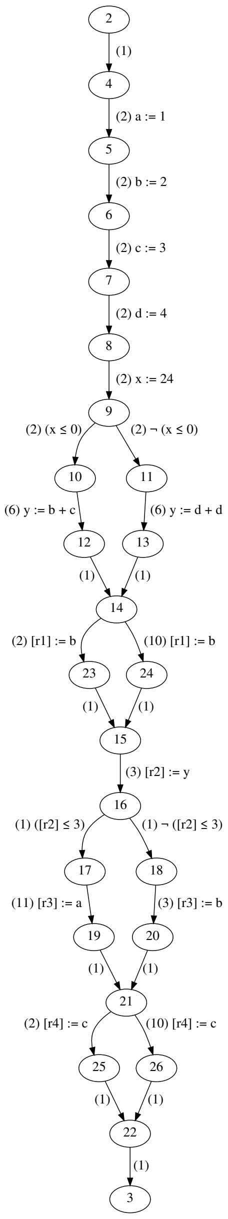
and erroneously conclude that the program has constant execution time (specifically: is independent of choices at $n = 9$).

The solution, of course, is to make different timing behavior due to micro-architectural state visible in the control flow graph, by modifying G at those nodes m at which execution may reach micro-architectural states with different timing behavior. These nodes can

[7] using, for example, the default timing cost model $\mathbb{1}$, or any timing cost model that assigns some constant timing cost to each variable access



(a) CFG *G*



(b) Micro-Architecture Aware CFG *G*′

be identified by inspection of G_α, by gathering for each G-node m all G_α-nodes

$$\left(m, \sigma_\mu^{1^\#}\right), \left(m, \sigma_\mu^{2^\#}\right), \ldots$$

and then checking if for any outgoing labeled edge, there may be different timing in the concrete semantics for any two $\sigma_\mu^i, \sigma_\mu^j$.

At such nodes m, I duplicate edges to fresh artificial nodes immediately behind m and its former successors. I call the resulting graph G' the micro-architecture aware control flow graph for G.

Consider Figure 13.4b at page 259, which shows the result G' of this transformation, together with an explicit timing cost model C'. There, a cache-miss is assumed to take 10 units of time, while a cache-hit takes 2 units[8]. At node 14, the read from `b` takes either 2 or 10 units of time, since `b` there might either be in the cache, or not.[9] Hence in G', node 14 has *two* artificial successors: the read from `b` takes either 2 or 10 units of time, since `b` there might either be in the cache, or not. On the other hand, node 15 still has only one successor, reached with timing cost $3 = 2 + 1$ (cache access plus register access), since I found that there the variable `y` is always in cache (by inspection of Figure 13.1b).

In G', I now have (as desired) that node 21 is in the backward slice of the exit node 3.

$$21 \in \left(\to_{\text{tscd}}^{G'[C']}\right)^* (\{3\})$$

Note that even if G is deterministic, G' usually is not. This is no problem, because I can still use the micro-architectural dependencies

[8] memory writes are assumed to always take 2 units of times, and register accesses take 1 unit of time

[9] In the timing cost model C, the cost $11 = 10 + 1$ that stems from one uncached variable access plus one register access is split into two edges. I need to do this because in my notion of graphs, there can be no multi-edges, and since I required cost models C to be *strictly* positive.

$\rightarrow^{G}_{\mu\mathrm{d}}$ (and data dependence ${\rightarrow_{\mathrm{data}}}^{G}$) from the original graph G, and only use G' for timing sensitive control dependence $\rightarrow^{G'[C']}_{\mathrm{tscd}}$.

Observation 13.5.1 (Soundness of Micro-Architectural Dependence for a Data Cache Micro-Architecture)**.** Assume the micro-architecture consisting of the fully associative data cache of size $c \in \mathbb{N}$ with least-recently used eviction strategy, as used in the examples of this chapter.[10] As in the examples, choose the abstractions

$$\alpha = \alpha_{\mathrm{cache}} \quad \text{and} \quad \alpha^{⏱} = \alpha^{\mathrm{use}(m)}_{\mathrm{incache}}$$

Let $G = (N, E)$ be any deterministic labeled control flow graph, $n^0 \in N$, and σ^0_μ be the initially empty cache. Also let σ^0_{M} be the initially empty, partial mapping from variables in G to values, and let $t^0 = 0$ be the starting time.

Assume – starting at n^0 — a linear sequence of "initialization" nodes n_y with outgoing edges labeled $y{:=}?$ for each variable y in G. Let $G' = (N', E')$ be the micro-architecture aware control flow graph for G, with associated timing cost model C'. Let $m \in N'$ be any non-initialization node, and let

$$S = \left(\rightarrow^{G'[C']}_{\mathrm{tscd}} \cup {\rightarrow_{\mathrm{data}}}^{G'} \cup \rightarrow^{G}_{\mu\mathrm{d}} \right)^{*} (\{m\})$$

be the micro-architectural, timing sensitive backwards slice of m.

Assume two *inputs* i_1, i_2 to G, i.e.: two mappings from variables y to some initial value $i_1(y)$ or $i_2(y)$, respectively. Also assume i_1, i_2 to coincide on all variables y for which the initialization node n_y is in the slice S.

Obtain G_1 from G by replacing all labels $y{:=}?$ at initialization nodes n_y by $y{:=}i_1(y)$, and obtain G_2 similarly.

[10] for the purpose of gathering experimental evidence, I chose $c \in \{4, 8\}$.

Then the timed, micro-architectural execution sequences for G_1 and G_2 (as defined by the full small-steps semantics in Definition 13.2.2) starting in $\left(n^0, \sigma_{\mathsf{M}}^0, t^0\right)$ coincide, after each removing all configurations $(n, \sigma_{\mathsf{M}}, t)$ for which $n \notin S$.

13.6 Arrays

In the preceding sections, I (implicitly) made two simplifying assumptions: every program variable resides in a distinct *memory block*, and the memory block of each variable access is statically known. This allowed me to represent abstract caches by lists of variables. The first assumption can easily be done away with: I merely need to respect static, non-injective mapping from variables to memory blocks (such that different variables can share a memory block), and then represent abstract caches by lists of *memory blocks*. Assuming a set $\mathbb{B} = \{b_0, b_1, \ldots\}$ of memory blocks, an abstract cache $\sigma'^{\#}_{\mu}$ is then just a list of memory blocks For example: $\sigma^{\#}_{\mu} = [\mathtt{b}_5, \mathtt{b}_1, \mathtt{b}_7, \mathtt{b}_8]$.

The simplest setting in which the second assumption no longer holds is one in which programs contain *array* accesses such as $l = \;\mathtt{r} := \mathtt{a}[\mathtt{i}]$ with statically unknown indices. Since arrays can span *multiple* memory blocks, the abstract micro-architectural transition from an abstract cache $\sigma'^{\#}_{\mu}$ along a label l is no longer deterministic. Consider, for example, the abstract cache $\sigma^{\#}_{\mu} = [\mathtt{b}_5, \mathtt{b}_1, \mathtt{b}_7, \mathtt{b}_8]$ and assume that each memory blocks a size of 64 bytes. Assume that the byte-array `a` is aligned with memory block b_1, and assume that `i` is statically known to be in the range $0 \ldots 255$. Then the control flow graph edge $n \xrightarrow{l}_G m$ will induce *four* abstract micro-architectural transitions:

$$\begin{array}{llll}
\left(n,\ \sigma^{\#}_{\mu}\right) \xrightarrow{l} (m,\ [\mathtt{b}_1, \mathtt{b}_5, \mathtt{b}_7, \mathtt{b}_8]) & \text{for } \mathtt{i} \in & 0 \ldots\ 63 & \text{in } \sigma_{\mathsf{M}} \\
\left(n,\ \sigma^{\#}_{\mu}\right) \xrightarrow{l} (m,\ [\mathtt{b}_2, \mathtt{b}_5, \mathtt{b}_1, \mathtt{b}_7]) & \text{for } \mathtt{i} \in & 64 \ldots 127 & \text{in } \sigma_{\mathsf{M}} \\
\left(n,\ \sigma^{\#}_{\mu}\right) \xrightarrow{l} (m,\ [\mathtt{b}_3, \mathtt{b}_5, \mathtt{b}_1, \mathtt{b}_7]) & \text{for } \mathtt{i} \in & 128 \ldots 191 & \text{in } \sigma_{\mathsf{M}} \\
\left(n,\ \sigma^{\#}_{\mu}\right) \xrightarrow{l} (m,\ [\mathtt{b}_4, \mathtt{b}_5, \mathtt{b}_1, \mathtt{b}_7]) & \text{for } \mathtt{i} \in & 192 \ldots 255 & \text{in } \sigma_{\mathsf{M}}
\end{array}$$

The framework for micro-architectural dependencies from still applies, but *as is* introduces unnecessary imprecision. The reason is that rule Suc on page 253 for the equivalence relation $\equiv$ is too strict, which results in the graph

$$G^{\equiv}_{\alpha,m,\alpha^{⏰}}$$

having a postdominance relation of too small a size.

Recall that rule Suc requires for two nodes (n, Σ_n) and (n, Σ'_n) in the graph $G_{\alpha,m,\alpha^{⏰}}$ that for all labels l, every l-transition from (n, Σ_n) is matched by a l-transition from (n, Σ'_n). In the example, this also requires l-transitions corresponding to $\texttt{i} \in 0 \ldots 127$ to be matched by l-transitions corresponding to, e.g., $\texttt{i} \in 192 \ldots 255$.

But it is enough to only require l-transitions corresponding to $\texttt{i} \in 0 \ldots 127$ to be matched by l-transitions corresponding to $\texttt{i} \in 0 \ldots 127$, and l-transitions corresponding to $\texttt{i} \in 64 \ldots 127$ to be matched by l-transitions corresponding to $\texttt{i} \in 64 \ldots 127$, etc.

In order to do this, I first modify the abstract micro-architectural semantics from Definition 13.2.3 on 240 to include in the transition labels also that part of the macro-architectural state σ_{M} on which the micro-architectural transition depends. Then rule Suc remains unmodified, except that now the variable l must be read to range over pairs $l_0, \sigma^{\#}_{\mathsf{M}}$ of control flow graph labels l_0 and abstractions of macro-architectural states.

Definition 13.6.1 (Abstract Micro-Architectural Semantics, Modified)**.** In addition to α as in Definition 13.2.3, overload the name α to also stand for a (abstraction) function

$$\alpha : \Sigma_{\mathsf{M}} \to \Sigma^{\#}_{\mathsf{M}}$$

from (concrete) macro-architectural states σ_{M} to some abstraction $\alpha(\sigma_{\mathsf{M}})$ in some set $\Sigma^{\#}_{\mathsf{M}}$ of abstract macro-architectural states. I demand that the abstraction is compatible with the micro-architectural label transition function l^{μ}, i.e. that: for any $\sigma^1_{\mathsf{M}}, \sigma^2_{\mathsf{M}}$ and $\sigma^1_{\mu}, \sigma^2_{\mu}$ with $\alpha\left(\sigma^1_{\mathsf{M}}\right) = \alpha\left(\sigma^2_{\mathsf{M}}\right)$ and $\alpha\left(\sigma^1_{\mu}\right) = \alpha\left(\sigma^2_{\mu}\right)$ I have

$$\alpha\left(l^{\mu}\left(\sigma^1_{\mathsf{M}},\ \sigma^1_{\mu}\right)\right) = \alpha\left(l^{\mu}\left(\sigma^2_{\mathsf{M}},\ \sigma^2_{\mu}\right)\right)$$

Then the modified abstract micro-architectural small-steps semantics of a control flow graph G is just

$$\frac{n \xrightarrow{l}_G m \qquad \alpha\left(\sigma_\mu\right) = \sigma_\mu^\# \qquad \sigma'_\mu = l^\mu\left(\sigma_\mathsf{M},\, \sigma_\mu\right) \qquad \alpha\left(\sigma'_\mu\right) = \sigma'^\#_\mu}{\left(n,\, \sigma_\mu^\#\right) \xrightarrow{l,\alpha(\sigma_\mathsf{M})} \left(m,\, \sigma'^\#_\mu\right)} \text{Label}$$

In the example, the relevant abstract macro-architectural states are $\{\text{“}\mathtt{i} \in 0 \ldots 63\text{”}, \ldots\} \subseteq \Sigma_\mathsf{M}^\#$, and I have

$$\begin{aligned}
\left(n,\, \sigma_\mu^\#\right) &\xrightarrow{l,\text{“}\mathtt{i}\in 0\ldots 63\text{”}} \left(m,\, [\mathtt{b}_1, \mathtt{b}_5, \mathtt{b}_7, \mathtt{b}_8]\right)\\
\left(n,\, \sigma_\mu^\#\right) &\xrightarrow{l,\text{“}\mathtt{i}\in 64\ldots 127\text{”}} \left(m,\, [\mathtt{b}_2, \mathtt{b}_5, \mathtt{b}_1, \mathtt{b}_7]\right)\\
\left(n,\, \sigma_\mu^\#\right) &\xrightarrow{l,\text{“}\mathtt{i}\in 128\ldots 191\text{”}} \left(m,\, [\mathtt{b}_3, \mathtt{b}_5, \mathtt{b}_1, \mathtt{b}_7]\right)\\
\left(n,\, \sigma_\mu^\#\right) &\xrightarrow{l,\text{“}\mathtt{i}\in 192\ldots 255\text{”}} \left(m,\, [\mathtt{b}_4, \mathtt{b}_5, \mathtt{b}_1, \mathtt{b}_7]\right)
\end{aligned}$$

Remark 13.6.1. In order to limit the size of resulting transition relation, in practice it is necessary to establish tight static bounds on index variables such as `i` in $\mathtt{r} := \mathtt{a}[\mathtt{i}]$ whenever possible.

Summary

- Micro-architectural dependencies $\rightarrow_{\mu d}$ expose timing channels that arise due to the micro-architecture of the executing CPU.
- They can be computed using nontermination insensitive slices, and an additional greatest fixed point computation that is driven by nontermination insensitive postdominance $\sqsupseteq_{\text{SINK}}$.

14 Cache Timing Attacks on AES256

Dazzle!

(Dazzle — Dota 2)

Cache timing attacks on implementations of cryptographic primitives allow an attacker to recover cryptographic keys and/or plain text messages by observing only the execution time of encryption or decryption operations. Even if (during an computation of the cryptographic operation) the flow of control is independent from keys and plain text[1], the execution time may depend on these due to the effects of data caches. Differences in execution time can then be used to infer keys (e.g., [Ber05; BM06]).

In this chapter, I employ micro-architectural dependencies to analyze cache-based timing channels in implementations of the AES256 block cipher. As expected, my analysis discovers timing channels in naive *substitution table* based implementations. In two more sophisticated implementations, my analysis can proofs the absence of cache based timing channels.

[1] i.e., even if these do not affect decisions at branch- and loop-predicates

14.1 AES256 Encryption

A crucial operation during AES256 encryption is the “S-Box” substitution step, which substitutes values in the current computation state by their value in a constant, publicly known substitution table. The most natural *default* implementation is by simple array lookup, as shown in Figure 14.6a. Here, `state` is an array with 16 entries that holds the current encryption state, i.e.: the encrypted message. The array `sbox` is the constant, publicly known substitution table with 256 entries. In this implementation, the execution time is affected by the current `state` (and hence: by plain text and key), because the value `r2` affects which part of the substitution table `sbox` is accessed. Since in common CPU micro-architectures the array `sbox` will span multiple memory blocks[2], the value `r2` in one iteration may influence whether the read of `r3` in a later iteration is served from the data cache or from the main memory, and hence the execution time. In the micro-architecture aware control flow graph, the exit node (i.e.: the total execution time) is timing sensitively control dependent on the node corresponding to line 4, which in turn is data dependent on the key and the plain text.

Additional timing dependencies, also due to micro-architectural dependencies, exists. For example, there exists a dependency chain from the exit node to the plain text input node via a micro-architectural dependency from the node corresponding to line 4 in Figure 14.6a to the node corresponding to line 5 in Figure 14.7a).

Figure 14.1a) shows the results of cache- and timing sensitive slicing w.r.t the exit node of the default AES256 implementation, for different assumed cache sizes. They were computed in my prototype implementation of micro-architectural dependencies in the Haskell programming language. The first column indicates the assumed cache size, in number of cache-lines. For simplicity, each scalar program variable was assumed to occupy a distinct memory block, while arrays (all of size 256, with 8-bit entries) were assumed to span exactly four

[2] i.e., regions of memory that are associated with the same cache line

memory blocks. Cache lines are assumed to fit exactly one memory block, effectively resulting in a cache line size of 64 bytes. The second column shows the size of the abstract micro-architectural transition graph G_α, measured in the number of nodes N_α. The third column shows the size of the micro-architectural dependency relation. The fourth column shows the size of the backward-slice (as described in Observation 13.5.1 on page 261) of the control flow graphs exit node. The next two column show execution time of the analysis, and the required amount of memory. The last column indicates whether the analysis could prove the absence of timing channels, i.e.: whether the slice contain *no* key or plain-text input node.

The given computation time includes the whole analysis: computation of G_α and the micro-architectural aware control flow graph G', as well as data-dependencies, timing-sensitive control dependencies, and micro-architectural dependencies, and slicing. The computation time is dominated by the computation of the graphs

$$G^{\equiv}_{\alpha,m,\alpha^{⏱}}$$

required for micro-architectural dependencies. All times in this chapter were measured on a high end "computation server" class PC with an Intel Xeon Gold 6230 CPU at 2.10 GHz base frequency with 512GB RAM.

For cache sizes of 12 cache lines or more, the analysis did not finish for this AES256 implementation. For 12 cache lines, computation exhausted the maximal memory (500GB). I expect that an analysis implementation more memory-efficient than my prototype Haskell implementation is possible, but since the size of the graph G_α appears to grow exponentially in the number of cache lines, I must expect this to extend the range of testable cache sizes *for this input program* only by a little.[3]

[3] But on the other hand, the phase of "exponential" growth of G_α may end within reach of a more efficient analysis-implementations. Just peek ahead to Figure 14.1b). There, the same Haskell analysis is run for a different input program (concretely: a different

Together, Figure 14.6, 14.7, 14.8, 14.9 , 14.10 and 14.11 form an implementation of AES256 encryption for one block of plain text data. Where applicable, the left subfigure shows the "naive" default implementation with an explicit "S-Box" substitution table `sbox`.

There are two common approaches to avoid cache timing channels in AES256 implementations. The first approach is to employ "precaching" of relevant cache lines. My Implementation of this approach is shown in the right hand sides of the aforementioned figures, and explained in Section 14.2. The second approach is to avoid table lookup depending on secret values altogether, by encoding the "S-Box" substitution in a constant time Boolean program (Figure 14.2 and Section 14.3).

AES256 implementation). But for that program, the graphs G_α are much smaller, *and* the growth appears to slow at 16 cache lines, and then stop at 32.

# c.l.	$\lvert N_\alpha \rvert$	$\lvert \mu\, \mathrm{d} \rvert$	$\lvert S \rvert$	time (s)	memory (kb)	passed
2	1132	8	695	1	30 968	✗
4	8349	39	695	3	35 040	✗
6	89378	132	695	47	268 488	✗
8	529225	324	695	396	2 097 396	✗
10	2627483	463	695	20955	62 865 540	✗
12	18444663	did not finish				

(a) Default AES265 Implementation

# c.l.	$\lvert N_\alpha \rvert$	$\lvert \mu\, \mathrm{d} \rvert$	$\lvert S \rvert$	time (s)	memory (kb)	passed
2	1189	21	703	1	39 128	✗
4	2901	66	717	3	36 152	✗
6	6320	85	427	4	36 952	✗
8	14570	50	62	5	65 780	✓
10	21136	44	62	7	84 184	✓
12	47593	96	72	30	229 676	✓
14	110128	67	74	103	981 228	✓
16	146792	101	58	166	1 814 764	✓
18	149650	45	54	134	2 077 944	✓
20	149696	18	44	90	1 919 220	✓
22	149712	18	40	60	1 627 332	✓
24	149718	18	40	60	1 921 264	✓
26	149724	18	40	60	1 646 780	✓
28	149736	18	40	60	2 101 312	✓
30	149748	18	40	60	1 688 752	✓
32	151009	18	40	60	1 961 124	✓
...						
100	151009	18	40	60	1 881 268	✓
200	151009	18	40	60	1 511 584	✓
500	151009	18	40	60	1 795 240	✓

(b) With *pre-caching*

Figure 14.1: Cache- and Timing Dependencies

14.2 Pre-Caching

For architectures with big data caches, pre-caching is easy: one simply pre-caches *all* memory blocks associated with program data at the start of the program. For very small data caches, on the other hand, pre-caching must be employed more strategically, with respect not only to the usage of cache-lines, but also to the actual cache size. I am not aware of any automatic process to do this. But micro-architectural and timing can automatically verify the correctness of manual pre-caching for a given cache size.

In Figure 14.6 and 14.7, I surrounded access to the `sbox` and the `skey` arrays[4] by loads that pre-cache the corresponding array, and then after the access establish a fixed position of all relevant memory blocks in the assumed LRU cache order. In Figure 14.10, the additional loads establish fixed cache-positions for the `state` memory blocks after the key-expansion. In Figure 14.11, the first additional loads force variables $\mathtt{w}_i$ near to the front of the LRU cache — which they are anyway, if the `if` branch is not taken. Similarly, the next two additional loads "emulate" the memory access taken in the first branch of the second `if` statement. The last additional load pre-caches access to `skey`.

As can be seen from Figure 14.1b), my analysis can automatically proof the absence of timing channels due to data caches, for micro-architecture with 8 or more lines of data cache. For this program, my analysis takes up to 166 seconds, and up to $\approx$ 2GB of memory. The reason that the analysis time decreases at 18 cache lines is that my implementation can soundly skip the computation of the set $\{\, n \mid n \rightarrow_{\mu\mathrm{d}} m \,\}$ for those nodes m at which memory accesses always take the same amount of time (Observation 14.2.1), as per the observation immediately below. But for larger caches, this can be true for more nodes m.

[4] the latter of which is holding the *expanded* round key

Observation 14.2.1. Let m be any node in the control flow graph G. If for all two nodes (m, Σ), (m, Σ') in the graph[5] G_α at control node m the rule

$$\frac{\sigma_\mu \in \Sigma \qquad \sigma'_\mu \in \Sigma' \qquad \begin{array}{c} m \xrightarrow{l}_G m' \\ (m,\, \sigma_{\mathsf{M}},\, \sigma_\mu,\, 0) \xrightarrow{l} \left(m,\, \sigma_{\mathsf{M}},\, \sigma'_\mu,\, \Delta t\right) \\ \left(m,\, \sigma_{\mathsf{M}},\, \sigma'_\mu,\, 0\right) \xrightarrow{l} \left(m,\, \sigma_{\mathsf{M}},\, \sigma'_\mu,\, \Delta t'\right) \end{array}}{\Delta t = \Delta t'}$$

is admissible, then

$$\{\, n \mid n \rightarrow_{\mu\mathrm{d}} m \,\} = \emptyset$$

Micro-Architectural dependencies do not only confirm that the employed pre-caching is sufficient, but the also helped me to conclude where it was necessary: I determined all additional loads in the pre-caching AES256 by manual inspection of the graph G_α as well as micro-architectural dependencies $\rightarrow_{\mu\mathrm{d}}$, starting from the default implementation and then iterating.

[5] Definition 13.3.1 on page 245

```
x0 := (state[0] & 128 ^ state[1] & 128 >> 1) ^ state[2] & ...
x1 := (state[0] & 64 << 1 ^ state[1] & 64) ^ state[2] & ...
...
y14 := x3 ^ x5
y13 := x0 ^ x6
...
t2 := y12 & y15
t3 := y3 & y6
...
z0 := t44 & y15
z1 := t37 & y6
...
t46 := z15 ^ z16
t47 := z10 ^ z11
...
s0 := t59 ^ t63
s6 := t56 ^ ~t62
state[0] := ((s0 & 128 ^ s1 & 128 >> 1) ^ s2 & 128 >> 2) ^ ...
state[1] := ((s0 & 64 << 1 ^ s1 & 64) ^ s2 & 64 >> 1) ^ ...
...
```

Figure 14.2: Constant Time: SUB^{ct}

14.3 Constant Time S-Box Substitution

In [BP10], the authors present a Boolean function that implements "S-Box" substitution purely by Boolean operations on the (bit-representation) of input. Figure 14.2 gives an idea. As expected, my analysis proofs the absence of timing channels if in the default implementation, the array based "S-Box" substitution is replaced by that of Figure 14.2. As can be seen from Figure 14.3 (with the same columns as before in Figure 14.1), this holds for *all* assumed cache sizes.

# c.l.	$\lvert N_\alpha \rvert$	$\lvert \mu\, \mathrm{d} \rvert$	$\lvert S \rvert$	time (s)	memory (kb)	passed
2	3717	7	51	3	220 344	✓
4	7062	27	60	5	285 832	✓
6	18946	102	67	15	221 372	✓
8	61399	156	64	57	275 576	✓
10	195541	155	64	204	785 580	✓
12	322495	146	63	304	1 321 208	✓
14	419209	141	62	371	1 640 572	✓
16	509983	137	60	431	1 957 040	✓
18	580357	132	59	491	2 115 768	✓
20	613723	125	58	492	2 178 216	✓
22	647089	125	58	531	2 378 896	✓
24	680455	125	58	542	2 521 324	✓
26	713821	125	58	605	2 334 844	✓
28	747187	125	58	598	2 816 188	✓
30	780553	125	58	685	2 819 240	✓

(c) Constant Time S-Box

Figure 14.3: Cache- and Timing Dependencies

14.4 Validation

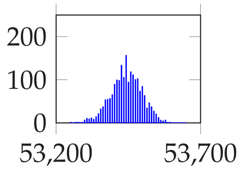
By Figure 14.1, I know that for the *pre-caching* implementation, execution time is independent from plain text and key for the assumed cache sizes starting at 8 cache lines. Since in AES256 encryption, key and plain-text are the programs *only* inputs, this implementation then is indeed a *constant time* implementation for 8 cache lines. Using an interpreter for the concrete micro-architectural, I validated these results for randomly chosen key and plain-text inputs. For such inputs, encryption took 46578 units of time in the assumed timing model.

For the implementation with constant time "S-Box" substitution (Figure 14.3), this holds even for 2 cache lines. There, encryption takes a constant 211189 units of time.

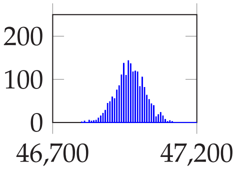
But to which extend is my analysis precise? By Figure 14.1, it cannot proof the default "naive" implementation constant-time for any cache size, and indeed it is not, as can be seen from Figure 14.4. It shows the execution time histograms for 2 to 12 cachelines, for 1000 random key and plain-text inputs. For the pre-caching implementation, my analysis cannot proof constant-time for 2 to 6 cache lines, but only for 8 and more cache lines. And indeed for cache-sizes 2 to 6, the pre-caching implementation is not constant time, as can be seen from Figure 14.5.

Summary

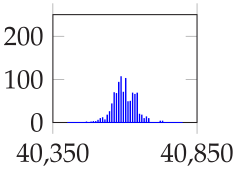
- Micro-architectural dependencies $\rightarrow_{\mu\mathrm{d}}$ can proof the absence of timing channels in AES256 implementations, under the assumption of a simple data cache micro-architectures.
- For one implementation, micro-architectural dependencies can only be computed for small assumed cache sizes.



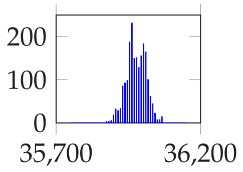
(a) 2 Cache Lines



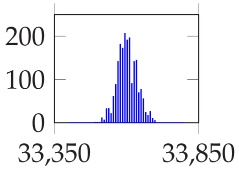
(b) 4 Cache Lines



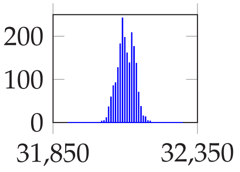
(c) 6 Cache Lines



(d) 8 Cache Lines

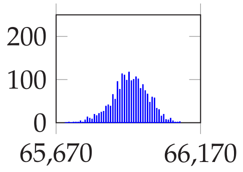


(e) 10 Cache Lines

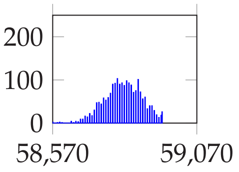


(f) 12 Cache Lines

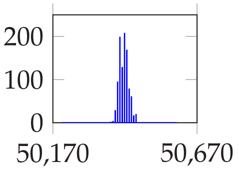
Figure 14.4: Execution Time: Default Implementation



(a) 2 Cache Lines



(b) 4 Cache Lines



(c) 6 Cache Lines

Figure 14.5: Execution Time: With *pre-caching*

```

for r1 : [0, 1 .. 15]
  r2 := state[r1]
  r3 := sbox[r2]
  state[r1] := r3
end

```

(a) Default: SUB^{def}

```
load sbox[0,64,128,192]
for r1 : [0, 1 .. 15]
  r2 := state[r1]
  r3 := sbox[r2]
  state[r1] := r3
end
load sbox[0,64,128,192]
```

(b) With *pre-caching*: SUB^{pc}

Figure 14.6: AES256: S-Box substitution

```

for r1 : [0, 1 .. 15]
  r2 := state[r1]
  r4 := (r1 + 0)
  r3 := skey[r4]
  r2 := (r2 ^ r3)
  state[r1] := r2
end

```

(a) Default: $\text{ADDROUND}_o^{\text{def}}$

```
load skey[0,64,128,192]
for r1 : [0, 1 .. 15]
  r2 := state[r1]
  r4 := (r1 + 0)
  r3 := skey[r4]
  r2 := (r2 ^ r3)
  state[r1] := r2
end
load skey[0,64,128,192]
```

(b) With *pre-caching*: $\text{ADDROUND}_o^{\text{pc}}$

Figure 14.7: AES256: Adding the Round Key

```
shiftRowsTmp := state[1]
state[1] := state[5]
state[5] := state[9]
state[9] := state[13]
state[13] := shiftRowsTmp
shiftRowsTmp := state[2]
state[2] := state[10]
state[10] := shiftRowsTmp
shiftRowsTmp := state[6]
state[6] := state[14]
state[14] := shiftRowsTmp
shiftRowsTmp := state[15]
state[15] := state[11]
state[11] := state[7]
state[7] := state[3]
state[3] := shiftRowsTmp
```

(a) SHIFT

Figure 14.8: AES256: Shifting Rows

```
MIX_0
MIX_4
MIX_8
MIX_12
```

(a) MIX

```
a0 := state[o + 0]
a1 := state[o + 1]
a2 := state[o + 2]
a3 := state[o + 3]
b0 := (a0 << 1) ^ (27 & ((a0 >> 7) * 255))
b1 := (a1 << 1) ^ (27 & ((a1 >> 7) * 255))
b2 := (a2 << 1) ^ (27 & ((a2 >> 7) * 255))
b3 := (a3 << 1) ^ (27 & ((a3 >> 7) * 255))
r0 := (((b0 ^ a1) ^ b1) ^ a2) ^ a3
r1 := (((a0 ^ b1) ^ a2) ^ b2) ^ a3
r2 := (((a0 ^ a1) ^ b2) ^ a3) ^ b3
r3 := (((a0 ^ b0) ^ a1) ^ a2) ^ b3
state[o + 0] := r0
state[o + 1] := r1
state[o + 2] := r2
state[o + 3] := r3
```

(b) MIX_o

Figure 14.9: AES256: Mixing Columns

```
EXPAND

ADDROUND_0
for r5 : [1, 2 .. 13]
   SUB
   SHIFT
   MIX
   ADDROUND_{r5 << 4}
end
SUB
SHIFT
ADDROUND_{14 << 4}
```

(a) Default

```
EXPAND
load state[0,64,128,192]
ADDROUND_0
for r5 : [1, 2 .. 13]
   SUB
   SHIFT
   MIX
   ADDROUND_{r5 << 4}
end
SUB
SHIFT
ADDROUND_{14 << 4}
```

(b) With *pre-caching*

Figure 14.10: AES256: Main Loop

```
n := 1
for i : [0, 1 .. 31]
  skey[i] := key[i]
end
for o : [32, 36 .. 236]
  w0 := skey[o−4]
  w1 := skey[o−3]
  w2 := skey[o−2]
  w3 := skey[o−1]
  if (o % 32 == 0) then
    rotateTmp := w0
    w0 := w1
    w1 := w2
    w2 := w3
    w3 := rotateTmp
    $\text{SUB}_4$
    w0 := w0 ^ rcon[n]

    n := n + 1
  end
  if (o % 32 == 16) then
    $\text{SUB}_4$



  end

  skey[o+0] := skey[o−32]^w0
  skey[o+1] := skey[o−31]^w1
  skey[o+2] := skey[o−30]^w2
  skey[o+3] := skey[o−29]^w3
end
```

(a) Default: $\text{EXPAND}^{\text{def}}$

```
n := 1
for i : [0, 1 .. 31]
  skey[i] := key[i]
end
for o : [32, 36 .. 236]
  w0 := skey[o−4]
  w1 := skey[o−3]
  w2 := skey[o−2]
  w3 := skey[o−1]
  if (o % 32 == 0) then
    rotateTmp := w0
    w0 := w1
    w1 := w2
    w2 := w3
    w3 := rotateTmp
    $\text{SUB}_4$
    w0 := w0 ^ rcon[n]
    load w1, w2, w3
    n := n + 1
  end
  if (o % 32 == 16) then
    $\text{SUB}_4$
  else
    load w0, w1, w2, w3
    load sbox[0,64,128,192]
  end
  load skey[0,64,128,192]
  skey[o+0] := skey[o−32]^w0
  skey[o+1] := skey[o−31]^w1
  skey[o+2] := skey[o−30]^w2
  skey[o+3] := skey[o−29]^w3
end
```

(b) With *pre-caching*: $\text{EXPAND}^{\text{pc}}$

.

Figure 14.11: AES256: Expanding the *Session Key*

15 Approximate Cache Dependencies

Das eigentliche Problem ist es ja nur, die richtigen Definitionen zu finden. Alles andere ist danach meist trivial.

(Joachim Cuntz (paraphrasiert) — Vorlesung Analysis III)

In the application of micro-architectural dependencies for simple data caches in Chapter 14, the computation of micro-architectural dependencies was practical for the two “constant time” AES256 implementations. But for the *naive* default AES256 implementation, the analysis was impractical for even moderate cache sizes (Figure 14.1a). The reason was that the size of the intermediate graph G_α appeared to grow exponentially in the assumed cache size.

To alleviate this problem, in this section I develop an approximation to micro-architectural dependencies $\rightarrow_{\mu\mathrm{d}}$ specifically for a simple cache architecture with least recently used eviction strategy. Since this approximation is best understood as an analogue of traditional data dependencies $\rightarrow_{\mathrm{data}}$ in labeled control flow graphs, I first review their definition in (Section 15.1). Then in Section 15.2, I introduce *local cache-cache* dependencies, which are those dependencies analog to the dependencies implicit in the def and use sets of standard data dependence. In Section 15.3, I describe dependencies from macro-architectural states to cache state (*local state-cache* dependencies). Then in Section 15.4, I describe the cache state analogue of a standard data dependence slice. Finally in Section 15.5, I show how this can be used to obtain an approximation to micro-architectural dependencies $\rightarrow_{\mu\mathrm{d}}$ for cache micro-architectures.

15.1 Data Dependence

Intuitively, a node m in a labeled control flow graph G is directly standard data dependent on a node n via variable x if a definition of variable x at node n may reach a use of variable x at node m. In this case, I write $n \overset{x}{\rightarrow}_{\text{data}} m$. Conversely, a use of x at node m is not directly data dependent on a definition of x at node n if on all control flow paths from n to m, this definition of x is "killed" by another definition of x at some other intermediate node.

Data dependence is typically computed "value-insensitively". If I understand the control flow graph to be an abstraction of all possible executions of the represented program, then each control flow graph node n abstractly represents all concrete configurations with control state n, but otherwise *arbitrary* variable states σ (e.g.: arbitrary mappings $\sigma : V \rightarrow \mathbb{N}$, for some set of program variables V, and a domain $\mathbb{N}$ of variable values). For standard data dependence, no attempts are made to consider only those concrete states σ at n which are possible in actual execution of the program, and no attempt is made to interpret the expressions used to define variables.[1] All that is considered are the corresponding use and def sets at control flow edges.[2]

Under this abstraction, data dependence can then simply be computed from the use and def sets by a standard (forward) data flow analysis. There implicitly, at each control flow edge labeled with l that is leaving a node n, every variable $x \in \text{def}(l)$ is considered to locally depend on every variable $y \in \text{def}(l)$. This is an approximation, since, for example, in the assignment `x := a * 0`, the variable `x` does semantically *not* depend on `a`. Another example is the assignment `x := a * b`, in which the local dependence of variable `x` on variable `a` as approximated by def and use sets does not hold semantically in programs for which $\sigma(\texttt{b}) = 0$ for all σ at node n.

[1] For example, an assignment `x := a * b` is treated no differently than `x := a + b`.
[2] For example: $\text{use}(\texttt{x := a * b}) = \text{use}(\texttt{x := a + b}) = \{\texttt{a,b}\}$, and $\text{def}(\texttt{x := a * b}) = \{\texttt{x}\}$.

15.2 Local Cache-Cache Dependencies

In order to obtain an approximation to micro-architectural dependencies $\to_{\mu\mathrm{d}}$ for simple data cache micro-architectures, I use a new notion of *local* cache-cache dependencies that can be thought of as an analogue of the local dependencies implicit in def and use sets for standard data dependence.

For reasonably precise results, I need for every control flow graph node n an approximation of all possible cache states at n. Similar to Section 13.6 earlier, a concrete cache state σ_μ for a data cache with k cache lines is just a list of memory blocks with up to k entries. The next-to-be-evicted memory appears at the end. For example, $[\mathtt{b}_5, \mathtt{b}_1, \mathtt{b}_7, \mathtt{b}_8]$ represents a cache with memory blocks $\mathtt{b}_i \in \mathbb{B}$, and block b_8 the next to be evicted.

Abstract Caches

In this chapter, I use a cache abstraction similar to that from [Doy+15]. An abstract cache state $\sigma^\#_\mu$ is a mapping from memory blocks to a set of possible positions in the concrete cache, i.e.: a mapping

$$\sigma^\#_\mu \ : \ \mathbb{B} \to 2^{\mathbb{K}}$$

where the set $\mathbb{K} = \{0, \dots, k-1\} \cup \{\infty\}$ is comprised of natural numbers $< k$, and the symbol ∞ which indicates that a corresponding memory block may be not in the cache. I write Σ_μ for the set of all such mappings $\sigma^\#_\mu$. A mapping from control flow graph nodes n to abstract caches $\sigma^\#_\mu(n)$ such that $\sigma^\#_\mu(n)$ soundly approximates all concrete caches possible at n is then available by abstract interpretation of control flow graph edges $n \xrightarrow{l}_G m$, and a forward data flow analysis.

In an abstract cache state $\sigma^{\#}_{\mu}$, the possible positions of two memory blocks are not necessarily disjunct. For example, the abstract cache $\sigma^{\#}_{\mu}$ with

b_0	b_1	b_2
$\{0,1\}$	$\{0,1\}$	$\{1,2,\infty\}$

indicates that memory block b_0 is guaranteed to be in the cache, and may be in the 0th or the 1st position in the concrete cache. The same holds for b_1, while memory block b_2 is indicated to be either in the 1st or 2nd position in the concrete cache, or not in the cache at all.

The set of all concrete caches represented by this abstract cache then is

$$[\mathtt{b}_0,\mathtt{b}_1,\mathtt{b}_2],\ [\mathtt{b}_0,\mathtt{b}_1],\ [\mathtt{b}_1,\mathtt{b}_0,\mathtt{b}_2],\ [\mathtt{b}_1,\mathtt{b}_0]$$

Note that by the constraint that any concrete cache position cannot contain two memory blocks, here the 0th and 1st position really must contain either b_0 or b_2, but never b_2.

In order to define notions of dependencies between memory blocks in abstract cache states, I do *not* defer to this ("precise") notion of concretization. Instead I use an approximate notion I call *pseudo*-concretization. The advantage is that with regard to pseudo-concretizations, these dependencies can be computed efficiently (as later described in Observation 15.2.1 and Observation 15.3.1).

By pseudo-concretization I mean the Cartesian product of the concretization of each individual memory block, i.e., for each abstract cache $\sigma^{\#}_{\mu}$ the set of functions

$$\sigma_{\text{pseudo}} \ : \ \mathbb{B} \rightarrow \mathbb{K}$$

that are compatible with $\sigma^{\#}_{\mu}$:

$$\sigma_{\text{pseudo}}(b) \ \in \ \sigma^{\#}_{\mu}(b)$$

for every $b \in \mathbb{B}$. For the previous example abstract cache $\sigma^{\#}_{\mu}$, this set is (with each row specifying one pseudo-concretization σ_{pseudo}):

b_0	b_1	b_2
0	0	1
0	0	2
0	0	∞
0	1	1
	...	
1	0	1
	...	
1	1	1
	...	

The LRU eviction strategy can be naturally extended to pseudo-concretizations σ_{pseudo}, by assigning to each control flow graph label l a function l^{μ} that maps macro-architectural state σ_{M} and a pseudo-cache state σ_{pseudo} to a successor pseudo-cache state σ'_{pseudo}.

Local Cache-Cache Dependencies

When slicing w.r.t standard data dependence, due to the local dependence of (for example) variable `x` on variable `y` in an assignment such as `x := y * 5`, variable `x` "inherits" all dependencies of variable `y`. The following definition formalizes a similar idea for caches.

Definition 15.2.1 (Local Cache-Cache Dependence)**.** Let $\sigma^{\#}_{\mu}$ be an abstract cache state, and l control flow graph edge label. Let $b \in \mathbb{B}$ be any memory block, and $b' \neq b$ be a memory block that may be accessed by label l. Then I say that b is locally cache-cache dependent on b' in $\sigma^{\#}_{\mu}$ via transition l if there exists pseudo-concretizations $\sigma^{1}_{\text{pseudo}}$ and $\sigma^{2}_{\text{pseudo}}$ of $\sigma^{\#}_{\mu}$ and some macro-architectural state σ_{M} with

$$\sigma^{1}_{\text{pseudo}}(b) \neq \sigma^{1}_{\text{pseudo}}(b') \quad \text{and} \quad \sigma^{2}_{\text{pseudo}}(b) \neq \sigma^{2}_{\text{pseudo}}(b')$$

such that

(a) $\sigma^{1}_{\text{pseudo}}$ and $\sigma^{2}_{\text{pseudo}}$ coincide on all memory blocks except b', and

(b) $\sigma^{1'}_{\text{pseudo}}(b) \neq \sigma^{2'}_{\text{pseudo}}(b)$,

where

$$\begin{aligned} \sigma^{1'}_{\text{pseudo}} &= l^{\mu}\left(\sigma_{\mathsf{M}}, \sigma^{1}_{\text{pseudo}}\right) \\ \sigma^{2'}_{\text{pseudo}} &= l^{\mu}\left(\sigma_{\mathsf{M}}, \sigma^{2}_{\text{pseudo}}\right) \end{aligned}$$

are the pseudo-concrete caches arising from transition along edge l.

In that case, I write

$$b' \xrightarrow{l}_{\text{local}\left[\sigma^{\#}_{\mu}\right]} b$$

For example, in the abstract cache $\sigma^{\#}_{\mu}$

b	b'	x
$\{1\}$	$\{0,2\}$	$\{0,2\}$

and a CFG edge labeled l = `r := b'` for a variable `b'` residing in block b', I do have $b' \xrightarrow{l}_{\text{local}\left[\sigma^{\#}_{\mu}\right]} b$. This is because in the (pseudo)-concrete cache with b' in position 0 and block x in position 2, the transition along l does *not* change the cache state (of any block), while in the pseudo-concrete cache with block b' in position 2 and block x in position 0, transition along l *does* change the cache state (specifically: of block b, but also of block x), by putting b' in position 0, and moving block b from position 1 to position 2.

On the other hand, in the abstract cache $\sigma^{\#}_{\mu}$

b	b'	$\dots$
$\{3\}$	$\{0,2\}$	$\dots$

I do *not* have $b' \xrightarrow{l}_{\mathrm{local}[\sigma^{\#}_{\mu}]} b$ for this label $l =$ `r := b'`.

I can equip the set $\mathbb{K}$ of abstract cache positions with an ordering $\leq$ by defining $a \leq \infty$ for all $a \in \mathbb{K}$, and use the order $\leq$ on natural numbers otherwise. With this, local cache-cache dependency can be simplified as follows:

Observation 15.2.1. Let l, b, b' and $\sigma^{\#}_{\mu}$ as in Definition 15.2.1. Let

$$b'_{\mathrm{min}} \quad = \quad \mathrm{min}_{a \in \sigma^{\#}_{\mu}(b')}\, a \qquad b'_{\mathrm{max}} \quad = \quad \mathrm{max}_{a \in \sigma^{\#}_{\mu}(b')}\, a$$

Then

$$b' \xrightarrow{l}_{\mathrm{local}[\sigma^{\#}_{\mu}]} b \quad \Longleftrightarrow \quad b'_{\mathrm{min}} < a < b'_{\mathrm{max}} \ \text{ for some } a \in \sigma^{\#}_{\mu}(b)$$

15.3 Local State-Cache Dependence

Local cache-cache dependencies from the previous section capture how dependencies from one memory block b' *transfer* to dependencies on other memory blocks b'. Now I formally define how dependencies of memory blocks initially arise from choices due to *macro-*architectural state.

These dependencies arise whenever at a labeled control flow graph edge $n \xrightarrow{l}_G m$ and an abstract cache state $\sigma^\#_\mu$, the macro-architectural state σ_{M} "chooses" which of multiple possible memory blocks b' is accessed. The canonical example is an array access `r2 := a[r1]` at which the value of register `r1` chooses which one of those several memory blocks over which the array `a` spans is actually accessed.[3] If then the resulting cache location of a memory block b depends on the choice made by the macro-architectural state σ_{M}, I say that there is a local state-cache dependence to b via l. The formal definition is:

Definition 15.3.1 (Local State-Cache Dependence)**.** Let $\sigma^\#_\mu$ be an abstract cache state, and l a control flow graph edge label. Let $b \in \mathbb{B}$ be any memory block. For each macro-architectural state σ_{M}, let $b'(\sigma_{\mathsf{M}})$ be the memory block accessed during transition along l from σ_{M}. Let B' be the set of all such memory blocks b'.

Then I say that in the abstract cache state $\sigma^\#_\mu$, block b is locally state-cache dependent (on the macro-architectural state) via l if there exists a pseudo-concretization σ_{pseudo} of $\sigma^\#_\mu$ and two macro-architectural states σ^1_{M} and σ^2_{M} relevant to b and σ_{pseudo} such that

$$\sigma^{1'}_{\text{pseudo}}(b) \neq \sigma^{2'}_{\text{pseudo}}(b)$$

[3] For simplicity, I assume that "dynamically" (i.e.: in the concrete semantics), for each label l at most one memory block is accessed. For example, I assume that no assignment like `r3 := a[r1] + b[r2]` occur, but instead only assignments like `r3 := a[r1]; r3 := r3 + b[r2]`.

where

$$\begin{array}{rcl} \sigma^{1'}_{\text{pseudo}} & = & l^{\mu}\left(\sigma^1_{\mathsf{M}},\, \sigma_{\text{pseudo}}\right) \\ \sigma^{2'}_{\text{pseudo}} & = & l^{\mu}\left(\sigma^2_{\mathsf{M}},\, \sigma_{\text{pseudo}}\right) \end{array}$$

are the pseudo-concrete caches arising from transition along edge l

By σ^1_{M} and σ^2_{M} being relevant to b and σ_{pseudo} I mean that block b must be accessed in the transition along l one of the macro-architectural states σ^i_{M}, or

1. $b'(\sigma^1_{\mathsf{M}}) = b$ or $b'(\sigma^2_{\mathsf{M}}) = b$, or
2. for both $i \in \{1,2\}$

$$\sigma_{\text{pseudo}}(b'(\sigma^i_{\mathsf{M}})) = \infty \ \text{ or } \ \sigma_{\text{pseudo}}(b'(\sigma^i_{\mathsf{M}})) \neq \sigma_{\text{pseudo}}(b)$$

For local state-cache dependencies involving b, l and $\sigma^{\#}_{\mu}$, I write

$$\xrightarrow{l}_{\text{choice}\left[\sigma^{\#}_{\mu}\right]} b$$

and later also $\quad n \xrightarrow{l}_{\text{choice}\left[\sigma^{\#}_{\mu}\right]} b \quad$ if $n \xrightarrow{l}_{G} m$ and $\sigma^{\#}_{\mu}$ appears at n.

Just as before for local cache-cache dependencies, local state-cache dependencies for a memory block b are determined by the possible cache positions of certain memory blocks b':

Observation 15.3.1. Let l, b, B' and $\sigma^{\#}_{\mu}$ as in Definition 15.3.1.

First, observe that if l never accesses any memory block b', or accesses the same memory block for every macro-architectural state σ_{M}, then b does not depend on any choice (of memory block).

Let for all $b' \in B'$:

$$b'_{\min} \;=\; \min_{a \in \sigma^{\#}_{\mu}(b')} a \qquad b'_{\max} \;=\; \max_{a \in \sigma^{\#}_{\mu}(b')} a$$

Then $\xrightarrow{l}_{\text{choice}[\sigma^{\#}_{\mu}]} b \iff$

$$\begin{array}{ll} & b^1_{\min} < a < b^2_{\max} \text{ for some } a \in \sigma^{\#}_{\mu}(b) \text{ and some } b^1, b^2 \in B' \text{ with } b^1 \neq b^2, \\ \text{or} & b \in B' \text{ and } B' \text{ is not a singleton set} \end{array}$$

15.4 Transitive Cache Dependencies

Local state-cache dependence from Section 15.3 “creates” dependencies of memory blocks on macro-architectural state, while local cache-cache dependence from Section 15.2 describes how dependencies from one memory block must be locally propagated to other memory blocks.

From the following rules, I obtain for each control flow graph node m and each memory block b at node m the set of all other nodes n on which b depends, in the sense that the macro-architectural state at node n may (transitively) influence the position of block b in the cache at node m. The resulting relation $\to_{\text{cache}}$ can be considered an analogue of the transitive closure of standard data dependence $\to$data.

Definition 15.4.1. For every control flow graph node n, let $\sigma_\mu^\#(n)$ be a sound approximation of all concrete cache states at n.[4]

A dependency of some memory block b is said to be killed in some abstract cache state $\sigma_\mu^\#$ if $\sigma_\mu^\#(b)$ is a singleton.

I write $\to_{\text{cache}}$ for the least solution $\to$ of the rule system

$$\frac{n \xrightarrow{l}_G m \qquad n \xrightarrow{l}_{\text{choice}\left[\sigma_\mu^\#(n)\right]} b}{n \to (m,b)}\ \text{CHOICE}$$

$$\frac{n \to (m,b) \qquad m \xrightarrow{l}_G m' \qquad \sigma_\mu^\#(m')(b) \text{ is not a singleton}}{n \to (m',b)}\ \text{TRANS}$$

$$\frac{\begin{array}{c} n \to (m,b') \\ m \xrightarrow{l}_G m' \qquad b' \xrightarrow{l}_{\text{local}\left[\sigma_\mu^\#(m)\right]} b \qquad \sigma_\mu^\#(m')(b) \text{ is not a singleton}\end{array}}{n \to (m',b)}\ \text{LOCAL}$$

[4] This is easily obtained by in standard data flow framework, by an abstract interpretation of the micro-architectural semantics for control flow graph edge labels l.

Then for memory block b and control flow graph nodes m and n, I say that at m, memory block b depends on n if

$$n \rightarrow_{\text{cache}} (m, b)$$

15.5 Approximate Cache Dependencies

I want to use transitive cache dependencies $n \rightarrow_{\text{cache}} (m, b)$ to obtain an approximation of micro-architectural dependencies as defined in Chapter 13. Transitive cache dependencies $n \rightarrow_{\text{cache}} (m, b)$ capture dependencies arising from macro-architectural choices at array accesses like `r2 := a[r1]`, but they do not capture influence arising from macro-architectural choices at conditional control flow graph nodes. In order to obtain a sound approximation, I hence need to respect (nontermination insensitive) control dependencies.

Definition 15.5.1. Let $\sigma^{\#}_{\mu}$ be an abstract cache state, and l control flow graph edge label. Let $b \in \mathbb{B}$ be any memory block. Then I say that b may be *modified* in $\sigma^{\#}_{\mu}$ by transition l if b may be accessed by label l, or if there exists a memory block b' that may be accessed by label l, and some cache positions

$$\begin{array}{rcl} a & \in & \sigma^{\#}_{\mu}(b) \text{ and} \\ a' & \in & \sigma^{\#}_{\mu}(b') \end{array}$$

with $a < a'$.

In that case, I write

$$\begin{array}{rl} & \xrightarrow{l}_{\text{mod}\left[\sigma^{\#}_{\mu}\right]} b \\ \text{and also} & n \xrightarrow{l}_{\text{mod}\left[\sigma^{\#}_{\mu}\right]} b \quad \text{if } n \xrightarrow{l}_{G} m \text{ and } \sigma^{\#}_{\mu} \text{ appears at } n. \end{array}$$

Definition 15.5.2. Let n, m be two nodes in control flow graph G. Let B be the set of cache lines b that may be accessed along control flow graph edges $m \xrightarrow{l}_{G} m'$ such that in $\sigma^{\#}_{\mu}(n)$, cache line b is not either guaranteed to be in the cache, or guaranteed to be *not* in the cache, i.e.:

$$\neg \left(\infty \notin \sigma^{\#}_{\mu}(n)(b) \ \vee \ \sigma^{\#}_{\mu}(n)(b) = \{\infty\} \right)$$

and write

$$G_m = G_{m \not\to}^{\to^* m}$$

Then I say that m is approximately cache micro-architecture dependent on n, and write $n \to^{\#}_{\mu\mathrm{d}} m$, if

$$\begin{array}{lll} & n \in \left(\to^{\mathrm{G_m}}_{\mathrm{nticd}}\right)^* (N') & \text{for } N' = \{\, n' \mid n' \to_{\mathrm{cache}} (m,b),\ \ b \in B \,\} \\ \text{or} & n \in \left(\to^{\mathrm{G_m}}_{\mathrm{nticd}}\right)^* (N') \setminus N' & \text{for } N' = \{\, n' \mid n' \xrightarrow{l}_{\mathrm{mod}\left[\sigma^{\#}_{\mu}(n')\right]} b,\ b \in B \,\} \end{array}$$

Remark 15.5.1. Note that since (micro-architectural) cache state can depend on macro-architectural choices, but macro-architectural choices cannot depend on cache-state, I need *not* consider an iterated slice

$$\left(\to^{\mathrm{G_m}}_{\mathrm{nticd}} \cup\ \to_{\mathrm{cache}}\right)^* (m)$$

or similar.

Approximate cache micro-architecture dependency really is a sound approximation of micro-architecture dependence for cache micro-architectures:

Observation 15.5.1.

$$n \to_{\mu\mathrm{d}} m \implies n \to^{\#}_{\mu\mathrm{d}} m$$

15.6 Improving the Precision

By rule TRANS for the computation of transitive cache dependencies $\rightarrow_{\text{cache}}$ in the previous section, a dependency $n \rightarrow (m, b)$ is only ever "killed" along a control flow graph edge $m \xrightarrow{l}_G m'$ if the approximation $\sigma^{\#}_{\mu}(m')(b)$ is a singleton set. But consider the following program, assuming that the array `a` spans 4 memory blocks, and also assume a cache size of $k = 4$.

```
r_a1 := a[r1]; r_x := x; r_y := y; r_a2 := a[r2]; r_u := u; r_v := v;
```

Assume that scalar variables `u,v,x,y` reside in distinct memory blocks. Then the choice[5] at the first access `r_a1 := a[r1];` to array `a` cannot possibly influence the position of any memory block corresponding to array `a` at the end of the program: Due to the accesses to the four scalar variables, the first choice is followed by four memory accesses to distinct memory blocks. Hence for a cache of size 4, the first choice must have been "evicted" by the end of the program. Only the second choice at `r_a2 := a[r2];` impacts the position of memory blocks corresponding to array `a` at the end of the program.

But the approximations $\sigma^{\#}_{\mu}(m)(a)$ of memory blocks a in the span of array `a` are never a singleton in between the first choice and the end of the program. Hence by the rules from the previous section, the dependency from the first choice is never killed.

In my prototype implementation, in order to obtain more precise transitive cache dependencies, I track for each dependency its "age", and then kill dependencies after traversal of control flow graph edges $m \xrightarrow{l}_G m'$ not only if the approximation of b at m' is a singleton, but also if since it's creation, the dependency must have been *evicted* by k accesses to other memory blocks that each must have pushed block b towards the back of the cache. Observation 15.5.1 still holds.

[5] due to the value of `r1`

15.7 Approximation in AES256 Implementations

In order to evaluate the loss of precision in the approximation Observation 15.5.1, I applied it to the three AES256 implementations from Chapter 14. The results are shown in Figure 15.1, 15.2 and 15.3.

As expected for the "naive" default Implementation, the approximation can report possible timing channels not only for cache sizes up to 10 cache lines, also for much bigger assumed cache sizes. The more precise analysis previously (in Figure 14.1 on page 271) could not be run for cache sizes bigger than 10 cache lines.

For the *pre-caching* implementation, the approximate analysis can proof absence of timing channels only for 10, 14 or more cache lines. The more precise analysis could show this also for 8 and 12 lines.

For the implementation with constant time "S-Box" computation, the approximate analysis can proof absence of timing channels for all assumed cache sizes.

The running time and memory requirements for all analysis runs were negligible for small cache sizes, but become more significant for high number of cache lines in the implementation with constant time "S-Box" computation (Figure 15.3).

# c.l.	$\lvert\mu^{\#}\,\mathrm{d}\rvert$	$\lvert S\rvert$	time (s)	memory (kb)	passed
2	29	695	1	34 812	✗
4	126	695	1	33 780	✗
6	558	695	1	38 964	✗
8	639	695	1	40 936	✗
10	617	695	1	35 836	✗
12	529	695	1	38 908	✗
14	507	695	1	42 928	✗
16	557	695	1	68 588	✗
18	557	695	1	80 824	✗
20	557	695	1	76 776	✗
22	311	695	1	75 764	✗
24	311	695	1	75 744	✗
26	311	695	1	76 780	✗
28	311	695	1	75 772	✗
30	311	695	1	94 152	✗
32	311	695	1	90 108	✗
			…		
100	251	695	1	97 224	✗
200	251	695	1	96 152	✗
500	251	695	2	94 256	✗
1000	251	695	2	95 176	✗
2000	251	695	3	90 052	✗

Figure 15.1: Default AES265 Implementation

# c.l.	$\lvert\mu^{\#}\,\mathrm{d}\rvert$	$\lvert S\rvert$	time (s)	memory (kb)	passed
2	63	703	1	37 864	✗
4	219	717	1	40 916	✗
6	292	717	1	39 956	✗
8	175	705	1	38 856	✗
10	260	73	1	40 928	✓
12	337	432	1	38 560	✗
14	304	79	1	41 956	✓
16	296	79	1	46 020	✓
18	316	83	1	42 924	✓
20	316	83	1	48 008	✓
22	32	40	1	44 004	✓
24	32	40	1	50 132	✓
26	32	40	1	44 156	✓
28	32	40	1	42 912	✓
30	32	40	1	48 012	✓
32	32	40	1	47 020	✓
			...		
100	32	40	1	51 180	✓
200	32	40	1	51 000	✓
500	32	40	1	51 232	✓
1000	32	40	1	50 996	✓
2000	32	40	1	50 140	✓

Figure 15.2: With *pre-caching*. Imprecision is highlighted.

# c.l.	$\lvert\mu^{\#}\,\mathrm{d}\rvert$	$\lvert S\rvert$	time (s)	memory (kb)	passed
2	17	52	3	206 760	✓
4	77	60	3	205 764	✓
6	260	67	3	230 316	✓
8	222	65	3	290 620	✓
10	208	64	3	254 976	✓
12	208	64	3	267 264	✓
14	206	63	3	226 104	✓
16	176	60	3	256 996	✓
18	176	60	3	272 384	✓
20	162	59	3	241 628	✓
22	148	58	3	247 804	✓
24	148	58	4	297 940	✓
26	148	58	4	289 680	✓
28	148	58	4	248 796	✓
30	148	58	4	249 816	✓
32	148	58	4	269 252	✓
			...		
100	148	58	15	1 995 740	✓
200	157	57	47	5 969 956	✓
500	157	57	103	5 969 836	✓
1000	157	57	194	5 962 748	✓
2000	157	57	376	5 962 720	✓

Figure 15.3: Constant Time S-Box

15.8 Related Work

Both the generic micro-architectural dependencies from Chapter 13, as well as (LRU data-cache specific) approximate cache dependencies from this chapter extend standard *dependency graphs* to realize a timing sensitive static information flow analysis sensitive to timing effects due to the (modeled) CPU micro-architecture. To the best of my knowledge, these two are the first such analyses based on dependency graphs.

The perhaps best-known static analysis for cache based timing channels is the *CacheAudit* system[Doy+15]. Based on approach of *counting* the set of possible observations (e.g., [KRB09]), *CacheAudit* also allows for the *quantification* of information leaks. With respect to timing leaks, *CacheAudit* counts the number of different possible execution times in a sound approximation of all possible execution paths. This approximation relies on a sound approximation not only of the possible cache state, but also of macro-architectural (e.g., heap and register) state. In order to obtain useful approximations of the possible execution times in all possible execution paths, the analysis must be able to statically bound the number of executions of each loop. In contrast, neither my generic micro-architectural dependencies, nor my approximate cache dependencies requires such a bound[6]. Two principal advantages of *CacheAudit* over my approaches is a) that it provided a quantification of information leakage for insecure program, and b) it can quantify information leakage not only for timing based attackers (i.e.: those who observe the total execution time), but also *access* based attackers (who monitor the access time of individual statements in *their own programs* running in parallel to the attacked program, in a shared cache environment), and *trace based* attackers who can monitor the sequence of cache hit and miss events. On the other hand, my approach provides not a quantification, but an *explanation* of timing leaks, in form

[6] Arguably, this is not a crucial advantage of my approach, since in the most relevant class of programs, i.e.: implementation of cryptographic programs, such bounds can usually be inferred or provided

of the involved micro-architectural dependencies (as well as standard control- and data-dependencies). In Chapter 14, I used these dependencies to derive a pre-caching AES256 implementation that did *not* require all involved memory blocks to be in the data cache at all times, but instead also works (i.e.: is without timing leaks) for smaller data cache sizes.

Aside from a LRU caches, the *CacheAudit* system also implements FIFO and *pseudo*-LRU replacement strategies. These also immediately fall into my generic micro-architectural framework, but would require new analogues of Observation 15.3.1 and Observation 15.2.1 for *approximate* cache dependencies.

Other approaches to static timing sensitive information flow analysis sensitive to cache effects are based on product programs resulting from self-composition (e.g., [Alm+16]), or symbolic execution ([Cha+19; Bro+19]). A short bibliography to approaches not based on static program analysis can be found, e.g, in [Doy+15], chapter 8.

Summary

- Micro-architectural dependencies for simple data cache micro-architectures can be approximated by an analogue of data dependencies.
- The approximation is reasonably precise, and allows for the analysis of large modeled cache sizes.

16 Timing Sensitivity in Concurrent Programs

Never tell me the odds!

(Han Solo — Star Wars)

In [Gif12] and [GS15], the authors present dependency graph based algorithms for static information flow control in concurrent programs. The underlying security criterion is **L**ow **S**ecurity **O**bservational **Determinism**, which requires *any* two executions to be completely indistinguishable by a "low" observer (who can only observe execution of certain "low" program points).

For many applications, this criterion is prohibitively strict, since even most simple programs *without any secret input at all* do not fulfill it. Consider the programs in Figure 16.1. Statements $\texttt{print}_l$ indicate (observable) output on a low channel l. The program on the left can have the observable trace $[42, 17]$ or $[17, 42]$ and hence is LSOD only if one assumes a deterministic scheduler, and not LSOD otherwise. The program on the right is not LSOD even for (some) deterministic schedulers (assume, for example, a round-robin scheduler).

The static checks in [Gif12] and [GS15] — if they succeed — guarantee LSOD for *any* scheduler. The authors also give static checks for a **R**elaxed notion of LSOD. Both notions imply **P**robabilistic **N**oninterference.

In [Bre+16] and [Bis+18b], we improved on both these static checks. There, we assume the scheduler to be

1. program-state independent, i.e.: a scheduling decision does not depend on, e.g., the state of variables, or the program counter of the threads to be scheduled.
2. stateless, i.e.: there is no other (scheduler-internal) state on which scheduling decision may depend

```
void main():
  fork thread_1();
  fork thread_2();
void thread_1():
  print_l(42);
void thread_2():
  print_l(17);
```

```
void main():
  l := read_l;
  fork thread_1();
  fork thread_2();
void thread_1():
  if (l <  10) {skip; skip; skip}
  print_l(42);
void thread_2():
  if (l >= 10) {skip; skip; skip}
  print_l(17);
```

Figure 16.1: Some Programs

3. probabilistic, in the sense that a scheduling decision may be random, with some underlying distribution.

We called such schedulers *probabilistic*. The canonical probabilistic scheduler is the *uniform* scheduler, which chooses one of the current n threads, each with probability $1/n$. Other probabilistic schedulers are possible. For example, given a *static* assignment of priority to (classes of) threads, a probabilistic scheduler may (after normalization) chose a given thread with probability p_i, where p_i is the probability of the class which the ith thread belongs to. All empirical results in this chapter are based on the uniform scheduler.

In this chapter, after reviewing probabilistic non-interference (Section 16.1) and a short general discussion of timing leaks (Section 16.2), I first review in Section 16.3 the improved relaxed LSOD criterion as described in [Bis+18b]. That section also includes formal definition of all auxiliary notions required in this chapter.

Then in Section 16.4, I describe the statistical test I used to empirically validate all of this chapters criteria for probabilistic noninterference. In Section 16.5, I explain how the criterion from Section 16.3 can be "less precise" than even Giffhorns original LSOD criterion from [Gif12] and [GS15].

Finally in Section 16.6, I propose a new criterion for probabilistic noninterference, based on *timing dependence* $\rightarrow_{\text{td}}$ as defined in Section 10.1.

16.1 Probabilistic Noninterference

Probabilistic Noninterference is based on the probability $P_i(t)$ of a given trace t to occur under input i to the given program. Given an otherwise deterministic program semantic, this probability is determined by the probabilistic scheduler. I sidestep any issues of actual *existence* of such a probability measure for a given program by simply demanding that for any given program, there exists a maximal length $k \in N$ such that for any input i, any trace t for input i is at most of length k. This will coincide with my empirical observations, which will be based on programs in a (concurrent) `For` language, i.e.: a language in which all loops will execute a number of times fixed at the beginning of the loop.[1] A less clumsy treatment of this issue can be found in [Bis+18a].

For a fixed probabilistic scheduler, a given program, and any input i to this program, let P_i be the corresponding probability measure. Let $T(i)$ be the set of possible traces for input i. Assume then a relation $\sim_L$ on traces t, such that $t \sim_L t'$ if t, t' are deemed (low)-observational equivalent, and write $[t]_L$ for the equivalence class of t. Assume a similar relation $\sim_L$ on inputs.

Definition 16.1.1 (Definition 6, [Bis+18b]). Let i, i' be inputs; let $\Theta = T(i) \cup T(i')$. Probabilistic Noninterference holds iff

$$i \sim_L i' \implies \forall t \in \Theta.\ P_i([t]_L) = P_{i'}([t]_L)$$

Remark 16.1.1. Requiring all traces of a program to be of maximal size k also sidesteps any issues of finding an appropriate relation $t \sim_L t'$. For example, in [GS15] and [Bis+18b], its definition is non-trivial in order to account for differences of two traces due to *infinite delay*. Also remember my development of a related notion in Section 6.6.

[1] i.e.: the language allows statements like `For h` C, which executes h iterations of C, where h is the value of the variable `h` upon reaching the statement. In a concurrent setting, such statements are implemented using a fresh *thread instance local* loop counter variable.

16.2 Observability of Internal Timing Leaks

All criteria in [Gif12; GS15; Bre+16] and [Bis+18b] assume a (externally!) timing *insensitive* notion of observation. For example, in the program on the left of Figure 16.1, the observer is assumed to really observe either $[42, 17]$ or $[17, 42]$. He is assumed to *not*, for example, observe timed traces such as [42⏱[3], 17⏱[4]] or [17⏱[3], 42⏱[4]].

Similarly in the program on the right of Figure 16.1. Assume, for a moment, that the read `l := read`$_l$ in line 2 was not from a L (public) channel, but from H (secret) channel. Note that here, if the observer *did* observe timed traces, he might easily conclude from observing, e.g, [42⏱[4], 17⏱[10]] that `l < 10` *must not* have been true, since in executions in which `l < 10` *does* hold, 42 can not be observed at time 4, but only as early as time 7 (assuming a simple timing model that takes one unit of time to evaluate each expression and statement). He would be able to directly observe an *external* timing leak.

But also when observing the *untimed* traces $[42, 17]$, an observer will (tentatively[2]) infer that `l < 10` was *less likely* to have held than not, since in the class of executions in which `l < 10`, the observation $[17, 42]$ is significantly *more* likely than the observation $[42, 17]$, while in the class of executions in which *not* `l < 10`, the observation $[17, 42]$ is significantly *less* likely than the observation $[42, 17]$.

Here, the delay of `print`$_l$`(42)` due to `l < 10` caused an *internal* timing leak, *made visible* to the server by the concurrent execution of `print`$_l$`(42)` and `print`$_l$`(17)`. The statement `print`$_l$`(17)` was also delayed (due to `l >= 10`), but the program would still have *not* been probabilistically noninterferent even without line this delay in 9.

In this chapter, I too assume a (externally!) timing *insensitive* notion of observation. But as I have just shown, I still have to be sensitive

[2] The observer would gain even more confidence in his conclusion that `l < 10`, if it possible for him to observe multiple executions of this program (with input a priori unknown to him, but known to him to be *constant*).

to *internal* timing, since this concurrency can make this visible even to timing-insensitive observers.

16.3 The RLSOD Criterion

In this section, I review the (improved) relaxed LSOD criterion from [Bis+18b]. There, we also described its derivation from the criteria in [GS15], which I do not repeat here.

In addition to control- and data-dependence, the criterion is based on the following new notions required in *concurrent* programs:

1. The **M**ay **H**appen in **P**arallel relation $m_1 \,\mathrm{MHP}\, m_2$.
2. Inter-Thread dependencies $n \xrightarrow{x}_{\text{inter}} m$, a form of concurrent data dependencies (along variable x), based on either MHP, or a **M**ay **H**appen in **B**efore relation MHB.
3. The *common dynamic ancestor* $\mathrm{cda}\,(m_1, m_2)$ of two nodes.

The May Happen in Parallel Relation

Consider a deterministic labeled control flow graph $G = (N, E)$ with labels l as in Section 13.2, together with a binary relation $\xrightarrow{\text{spawn}}$ on N of *spawn edges*. For simplicity, assume G to consist of a numbered set of disconnected parts called *threads*, and each jth thread with an (entry) node n^j, i.e. a node n^j such that all nodes m in thread j are reachable from n^j, but with no G-predecessors. Also assume that G consists of *trivial* control sinks only[3], and assume that spawn-edges $n \xrightarrow{\text{spawn}} m$ only enter entry nodes $m = n^j$ for some thread (number) j. I call the $0th$ thread with entry node n^0 the *main* thread.

Aside from the global variable state σ and *thread instance local* variable state σ_ι, I also assume a global state i for *input*-channels. Labels $a \in A$ then also include operations that operate on input- and output-channels. For example: labels for `read` statements that consume from

[3] such that $\rightarrow^G_{\text{nticd}}$ alone is an appropriate notion of control-dependence, and $\rightarrow_{\text{ntiod}}$ is not needed

i the next value available at some channel (yielding i'), and labels for `print` statements.

The single-threaded small step semantics then is

$$\frac{n \xrightarrow{a}_G m \qquad (\sigma',\, \sigma'_\iota,\, i') = a^{\mathsf{M}}\,(\sigma,\, \sigma_\iota,\, i)}{(n,\, \sigma,\, \sigma_\iota,\, i) \xrightarrow{a} (m,\, \sigma',\, \sigma'_\iota,\, i')}\,\text{State}$$

$$\frac{n \xrightarrow{g}_G m \qquad g^{\mathsf{M}}\,(\sigma,\, \sigma_\iota) = \mathbf{true}}{(n,\, \sigma,\, \sigma_\iota,\, i) \xrightarrow{g} (m,\, \sigma,\, \sigma_\iota,\, i)}\,\text{Guard}$$

while the full concurrent semantics then is

$$\frac{(n_\iota,\, \sigma,\, \sigma_\iota,\, i) \xrightarrow{l} (m_\iota,\, \sigma',\, \sigma'_\iota,\, i')}{\begin{array}{l}([(n_1,\sigma_1), \ldots, (n_\iota,\sigma_\iota), \qquad\qquad\qquad\qquad (n_{\iota+1},\sigma_{\iota+1}), \ldots, (n_k,\sigma_k)],\, \sigma,\, i) \\ \xrightarrow{\iota,l}([(n_1,\sigma_1), \ldots, (m_\iota,\sigma'_\iota), (n'_1,\epsilon), \ldots, (n'_K,\epsilon), (n_{\iota+1},\sigma_{\iota+1}), \ldots, (n_k,\sigma_k)],\, \sigma', i')\end{array}}$$

where $n'_1, \ldots, n'_K$ are the nodes n'_j such that $n_\iota \xrightarrow{\text{spawn}} n'_j$, and ϵ is some fixed initial thread-instance local state.

A configuration $\kappa = ([(n_1,\sigma_1), \ldots, (n_k,\sigma_k)],\, \sigma)$ is said to be at n if $n = n_\iota$ for some ι. A configuration may contain multiple *instances* of a thread j, i.e.: multiple nodes n_ι such that n_ι is in the jth disconnected part of G.

Given an initial state σ^0 and input i^0, a node n *may happen in parallel* to node m if there is a sequence

$$\left([n^0], \sigma^0,\, i^0\right) \xrightarrow{\text{\textvisiblespace}} \ldots \;\ldots \xrightarrow{\text{\textvisiblespace}} ([(n_1,\sigma_1), \ldots, (n_k,\sigma_1)],\, \sigma,\, i)$$

such that $n = n_\iota, m = n_{\iota'}$ with $\iota \neq \iota'$. This notion is symmetric.

I assume some sound, symmetric approximation MHP of this notion, i.e. a symmetric relation MHP such that if n *may happen in parallel* m, then $n\,\mathrm{MHP}\,m$.

Remark 16.3.1. Lacking a notion of *procedures*, this notion of MHP is (vacuously) not calling-context sensitive. It is also not thread-context sensitive, i.e.: it does not differentiate between two occurrences of a single node n in some configuration κ *based on the sequences of spawn nodes used to insert these occurrences* into κ. This information would be readily available at configurations if my semantics was based, for example, on *execution trees*[Gaw+11]. It can also be approximated by *thread invocation analysis*[Gif12].

Also, if I used execution trees, I could define a more refined *may concurrently happen before* relation. Intuitively, a node n *may concurrently happen before* m if a configuration at n may happen before a configuration at m, with m occurring in a "different" thread instance than that of the occurrence of n, *unless* the thread instance of m was (transitively) spawned by the thread instance of n (and after n).

This is difficult to define in my semantics. I don't want do consider a node n to concurrently happen before m only because in some sequence is first at n and then at m, because this may also be the case simply because m follows n in a single thread instance.

But note that since my semantics lacks any form of *synchronization*, my *may happen in parallel* notion approximates a proper *may concurrently happen before* notion, since if n *may concurrently happen before* m, then certainly also n *may happen in parallel* to node m. Just consider the execution sequence in which the thread instance of n remains at n, and only nodes from other thread instances are chosen to proceed, until m is reached from these other thread instances.

Inter-Thread Dependence

Intuitively, the value of a variable x defined at node n may be read at a node m in a thread instance different than that of n, if n may concurrently happen before m. As just argued, I can substitute *may concurrently happen before* by *may happen in parallel*, and use its static approximation MHP.

Definition 16.3.1. For nodes n, m,

$$\begin{array}{rlcl} & n \xrightarrow{x}_{\text{inter}} m & \iff & x \in \text{def}(n) \cap \text{use}(m) \text{ and } n \,\text{MHP}\, m \\ \text{and} & n \rightarrow_{\text{inter}} m & \iff & \exists x . n \xrightarrow{x}_{\text{inter}} m \end{array}$$

Inter-thread dependence $n \xrightarrow{x}_{\text{inter}} m$ does *not* account for the flow of data from a definition at node n to those nodes m that are (transitively) spawned from the thread instance in which n occured. An example is the definition `l := read`$_l$`;` of variable `l` in the right hand side of Figure 16.1 in line 2, to its use **`if`** `(l < 10)` in line 6.

In order to account for such flows, I need — in this chapter — standard data dependence $\rightarrow_{\text{data}}$ to include flow along spawn-edges $\xrightarrow{\text{spawn}}$, i.e.: for $G = (N, E)$, I use the data dependence relation for

$$G^{\text{spawn}} := \left(N, E \cup \xrightarrow{\text{spawn}}\right)$$

In the standard data-flow framework used to compute $\rightarrow_{\text{data}}{}^{G^{\text{spawn}}}$, then, definitions flow along edges $n \xrightarrow{\text{spawn}} m$ unmodified (i.e.: the transformer is the identity function).

Remark 16.3.2. Inter-Thread dependence $n \xrightarrow{x}_{\text{inter}} m$ is called *interference dependence* in [Kri98] and [Gif12].

Common Dynamic Ancestors

In [Bre+16] and [Bis+18b], we introduced the notion of **c**ommon **d**ynamic **a**ncestors of two nodes m_1, m_2. A motivating example is shown in Figure 16.2.

Here — somewhat similar to Figure 16.1 (right) — the observable statements `print`$_l$`(17)` and `print`$_l$`(42)` are delayed by the loop

```
 1  void main():
 2    h := read_h;
 3    for h { skip; }
 4    skip;
 5    fork thread_1();
 6    fork thread_2();
 7  void thread_1():
 8    print_l(42);
 9  void thread_2():
10    print_l(17);
```

Figure 16.2: The need for *common dynamic ancestors*

`for h { skip; }` at line 3, which executes `h` skip statements. A (externally) timing sensitive observer would learn the input to `h` by the execution time of the `print` statements.

What its *not* similar to Figure 16.1 (right) is that here in Figure 16.2, both `print`$_l$`(17)` and `print`$_l$`(42)` are delayed *by the same amount of time*. A (externally) timing insensitive observer will observe either $[42, 17]$ or $[17, 42]$, but neither observation will allow him to infer anything about the secret input value `h`. Even by repeatedly observing executions of this program, he will observe $[42, 17]$ roughly 2/3 of the time[4], and $[17, 42]$ roughly 1/3 of the time, *independent* of the value of `h`. In fact, after the delay at line 3, any execution of this program must first pass the `skip` statement at line 4 before reaching any of the two `print` statements that otherwise could have made the delay visible externally. The `skip` statement is a *common dynamic ancestor* of the two `print` statements.

Definition 16.3.2 (Common dynamic ancestor, [Bis+18b]). Let $n, m, c \in N$ be nodes in $G = (N, E)$, and n^0 the entry node of the main thread. Remember that G^{spawn} is the graph G together with $\xrightarrow{\text{spawn}}$ edges.

[4] since `thread_1` is forked first

1. c is a common dominator for m_1, m_2, written $c \sqsupseteq_{\text{CDOM}} (m_1, m_2)$, if c dominates both m_1 and m_2 in G^{spawn}, i.e.: if

$$c \sqsupseteq^{G^{\text{spawn}}}_{\text{DOM}} m_1 \text{ and } c \sqsupseteq^{G^{\text{spawn}}}_{\text{DOM}} m_2$$

2. c is a common dynamic ancestor for m_1, m_2, written $c \sqsupseteq_{\text{CDA}} (m_1, m_2)$, if

$$\begin{array}{ll} & c \sqsupseteq_{\text{CDOM}} (m_1, m_2) \\ \text{and neither} & c \,\text{MHP}\, m_1 \\ \text{nor} & c \,\text{MHP}\, m_2 \end{array}$$

3. If $c \sqsupseteq_{\text{CDA}} (m_1, m_2)$ and $\forall c' \sqsupseteq_{\text{CDA}} (m_1, m_2) \,.\, c' \sqsupseteq^{G^{\text{spawn}}}_{\text{DOM}} c$, then c is called an *immediate* common dynamic ancestor. I then write $c = \text{icda}\,(m_1, m_2)$.

Classification

Based on these three notions specific for concurrent programs, and existing notions of program dependence, in [Bis+18b] we then defined an improved, relaxed LSOD criterion for probabilistic noninterference. In the context of this chapter, I define the concurrent program dependence graph to be

$$\rightarrow_{\text{cpdg}} = \rightarrow^{G'}_{\text{nticd}} \cup \rightarrow_{\text{data}}{}^{G^{\text{spawn}}} \cup \rightarrow_{\text{inter}} \cup \xrightarrow{\text{spawn}}$$

Here, I treat spawn edges $m \xrightarrow{\text{spawn}} m'$ as dependencies in order to propagate control-dependencies $n \rightarrow^{G}_{\text{nticd}} m$ from m to m', since if m is control-dependent on n, then n also decides whether the thread starting in m' executes. In order to make sure that in fact then *all* nodes in the thread starting in m' are dependent on n, I compute control dependence not in the graph G, but in the graph G' obtained from G by adding an edge from each start node n^j of the jth thread to each *exit node* in thread j.

Then, given some information flow lattice $\mathcal{L}$, we defined the *classification* of nodes in G with regard to the (user provided) information flow specification ucl (the "user classification") to be the *least* solution of the rule system RLSOD, as follows:

Definition 16.3.3.

$$\frac{m \in I}{\mathrm{cl}\,(m) \sqsupseteq \mathrm{ucl}\,(m)}\ \text{INPUT} \qquad \frac{n \to_{\mathrm{cpdg}} \ m}{\mathrm{cl}\,(m) \sqsupseteq \mathrm{cl}\,(n)}\ \text{CPDG}$$

$$\frac{c = \mathrm{icda}\,(m_1, m_2) \qquad \begin{array}{c} m_1\, \mathrm{MHP}\, m_2 \\ c \to^*_{G^{\mathrm{spawn}}} n \ \to^*_{G^{\mathrm{spawn}}} m_i \end{array} \qquad i \in \{1,2\}}{\mathrm{cl}\,(m_i) \sqsupseteq \mathrm{cl}\,(n)}\ \text{CDA}$$

The specification ucl is given as a *partial* map from nodes to $\mathcal{L}$. The domain of ucl is assumed to be partitioned into sets of *input* nodes I and of *output* nodes O. Nodes m in the domain of ucl are specified to be observable at all levels $l \sqsupseteq \mathrm{ucl}\,(m)$. Specifically for the lattice $\mathcal{L}_2 = \mathrm{L} \sqsubset \mathrm{H}$, nodes m with $\mathrm{ucl}\,(m) = \mathrm{L}$ are observable at *all* levels, while nodes m with $\mathrm{ucl}\,(m) = \mathrm{H}$ are observable only at level H.

A program is then deemed secure at level $l \in \mathcal{L}$ by the RLSOD criterion (with respect to the specification ucl) if it admits the criterion rule

$$\frac{m \in I \cup O \qquad \mathrm{ucl}\,(m) \sqsubseteq l}{\mathrm{cl}\,(m) \sqsubseteq l}\ \mathsf{RLSOD}$$

Remark 16.3.3. If in the rule system RLSOD we instead use the subset-lattice on nodes N and the specification $\mathrm{ucl}^N\,(m) = \{m\}$, then for each m the solution $\mathrm{cl}\,(m)$ is a form of backward slice of m.

16.3.1 Observations

In order to connect judgments of the rule RLSOD to probabilistic non-interference, I need make concrete the notion $\sim_{\mathrm{L}}$ of equivalence of *traces*, and the notion $i \sim_{\mathrm{L}} i'$ of equivalence of *inputs*.

Here, an input i is a map from a fixed set of input-channels c^I to *streams* of values. I also assume a fixed set of output-channels c^O.

I extend specifications ucl to channels, i.e.: I demand that ucl maps all input-channels c^I and all output-channels c^O to some security level $l \in \mathcal{L}$.

Two inputs l, l' then are l-equivalent, and I write $i \sim_l i'$ iff i and i' coincide on input-channels classified $\sqsubseteq l$, i.e. iff

$$i\left(c^I\right) \;=\; i'\left(c^I\right)$$

for all input-channels c^I such that $\operatorname{ucl}\left(c^I\right) = l'$ for some $l' \sqsubseteq l$.

In need to define observations on execution-traces t, i.e.: observations of sequences

$$\ldots \;\overset{\llcorner\!\lrcorner}{\Rightarrow}\; ([\ldots,(n_\iota,\sigma_\iota),\ldots],\, \sigma,\, i) \xrightarrow{\iota,\lambda} ([\ldots,(n'_\iota,\sigma'_\iota),\ldots],\, \sigma',\, i') \;\overset{\llcorner\!\lrcorner}{\Rightarrow} \ldots$$

in the full concurrent semantics. I demand that the specification ucl for nodes is undefined for nodes n,[5] *unless* n has an outgoing control flow graph edge $n \xrightarrow{\lambda} m$ with λ being either an *input*-statement or an *output*-statement, i.e. a label of the form:

$$\begin{array}{rlll} \texttt{x := read}_{c^I}\texttt{;} & \text{on channel } c = c^I & \text{with } \operatorname{def}(\lambda) = \{x\}, \operatorname{use}(\lambda) = \emptyset \\ \text{or } \texttt{print}_{c^O}\texttt{(}x\texttt{);} & \text{on channel } c = c^O & \text{with } \operatorname{def}(\lambda) = \emptyset, \operatorname{use}(\lambda) = \{x\} \end{array}$$

In this case, I demand $\operatorname{ucl}(n) = \operatorname{ucl}(c)$.

[5] i.e.: n is unobservable for *any* observer

The l-observable subtrace t_l of t then is the sequence

$$\ldots \;\overset{\sqcup}{\rightarrow}\; (\sigma_{\text{use}},\, n_\iota,\, \lambda,\, \sigma'_{\text{def}}) \overset{\sqcup}{\rightarrow} \ldots$$

containing only those configurations for which $\text{ucl}\,(n_\iota) \sqsubseteq l$. Here, σ_{use} and σ'_{def} are suitable projections of $(\sigma_\iota,\, \sigma)$ to any variables $\text{use}\,(\lambda)$ printed via label λ, and of $(\sigma'_\iota,\, \sigma')$ to the set $\text{def}\,(l)$ of any variables read from input i via label λ, respectively.

Writing $t \sim_l t'$ whenever $t_l = t'_l$, we showed in [Bis+18b] that program deemed secure with respect to a specification ucl are probabilistically noninterferent.

Theorem 16.3.1 (Corollary 1 in [Bis+18b])**.** Let G be a program's labeled control flow graph, and ucl a specification G in the security lattice $\mathcal{L} = \text{L} \sqsubset \text{H}$. Then if the least solution cl of rule system RLSOD admits the rule RLSOD, probabilistic noninterference holds for G.

Consider again the program on the left of Figure 16.1. It is *not* low-security observational deterministic, but it is probabilistically noninterferent at level L. This is trivially so, since there is no node classified H. The same holds for the program on the right of Figure 16.1.

For Figure 16.2, the immediate common ancestor of the two `print` statements is the skip statement in line 4. But no node between this statement and the print statements is (transitively) dependent on the H input in line 2. Hence the program passes the RLSOD criterion, and is probabilistically noninterferent.

16.4 A Statistical Test for Probabilistic Noninterference

In order to facilitate the development of improvements to the RLSOD classification rules, I implemented an ad hoc statistical test for probabilistic noninterference. It consists of

1. A random program generator for a minimal concurrent language For.
2. A compiler from For abstract syntax trees into control flow graphs G and spawn edges $\xrightarrow{\text{spawn}}$.
3. An ad hoc statistical test that for two given inputs i, i' attempts to determine whether i, i' form a counter-example to the supposition that G is probabilistically noninterferent, by repeatedly *executing* G under inputs i and i' until such a determination can be made with confidence.

The language of For statements is made up from

- bounded integer valued program variables, which are either *global* or *thread local*, and have names in Var
- arithmetic expressions $\mathrm{E}_{\mathrm{Arithmetic}}$ (over program variables) $\mathrm{E}_{\mathrm{Boolean}}$ (over arithmetic expressions), with integer constants from Int.
- (static) thread identifiers T, with a designated main thread main.

Statements `For` n c implement loops with constant number of iterations, while the number of iterations of statements `For` x c is determined by the value of variable x immediately before the loop-statements execution. A For-*program* P then is a map $P : \mathrm{T} \rightarrow$ For from thread identifiers to For-commands.

The compiler from For-programs P to labeled control flow graphs implements a immediate, non-optimizing, syntax-tree directed translation. Loops `For` n c and `For` x c are both implemented as control flow graph cycles, using fresh thread-local variables as loop counters.

The interpreter for control flow graphs is a direct implementation of the concurrent semantic from Section 16.3.

Given a labeled control flow graph G (obtained from a For-program) and two L-equivalent inputs i, i', the ad hoc statistical tests must determine (with high confidence) whether inputs i, i' are a counterexample to the supposition that probabilistic noninterference holds for this program, i.e. whether or not it holds that

$$\forall t \in \Theta.\ P_i([t]_{\mathrm{L}}) = P_{i'}([t]_{\mathrm{L}}) \tag{16.1}$$

for the set $\Theta = T(i) \cup T(i')$ of traces possible under i, i'.

In order to be applicable to automated tests of (many) randomly generated programs, the statistical test must be *two-sided*. By this I mean that, given a set of $n \in \mathbb{N}$ randomly sampled executions of the program under input i, as well as a set of n randomly sampled executions of the program under input i', the test would ideally

1. Determine (with high confidence) that Equation 16.1 holds, i.e.: *reject* the null hypothesis

$$H_0^{\neq} \quad : \quad P_i \neq P_{i'}$$

2. *or* determine (with high confidence) that Equation 16.1 does *not* hold, i.e.: *reject* the null hypothesis

$$H_0^{=} \quad : \quad P_i = P_{i'}$$

3. *or*, if it can do neither, "request" more samples (i.e.: more executions of the program under both i and i') until it can.

But *for principal reasons*, I cannot expect a statistical test to ever reject $H_0^{\neq}$, even if I observe minimal (or even no) differences in the observed *empirical* distributions $\hat{P}_i$ and $\hat{P}_{i'}$, because such an observation can always *also* be explained equally well by two actual underlying distributions P_i and $P_{i'}$ that *do* differ, but only by some infinitesimal amount $\epsilon > 0$. *A nonsignificant difference must not be confused with significant homogeneity*[Wel10]. Hence all I can hope to do is to reject a modified null-hypothesis

1. $H_0^{\geq\epsilon}$: P_i differs from $P_{i'}$ by at least ϵ, in some metric

Remark 16.4.1. Although presumably possible, I did *not* use a modified *dual* null hypothesis in the second test, i.e.: I did not need to replace the null-hypothesis $H_0^{=}$ by some null-hypothesis

2. $H_0^{<\epsilon}$: P_i differs from $P_{i'}$ by less than ϵ, in some metric

In theory, keeping $H_0^{=}$ has the disadvantage that together with $H_0^{\geq\epsilon}$, it "logically consistent" to *simultaneously* reject $H_0^{=}$ and $H_0^{\geq\epsilon}$. In practice, this rarely a problem if I choose a very small epsilon, and require very high confidence $(1-\alpha)$.

I treat P_i (and similarly: $P_{i'}$) as a *multinomial* distribution over all possible observations of the program under inputs i, i' (i.e: over all possible equivalence classes $[t]_L \in \Theta/\sim_L$), represented by a vector

$$P_i = (p_{i,1}, \ldots, p_{i,j}, \ldots)$$

with $\sum_j p_{i,j} = 1$, and $p_{i,j}$ being the probability of the jth equivalence class $[t]_L$ of L-equivalent traces[6] under input i.

The standard approach to reject $H_0^{=}$ in this setting is (some variant of) Pearson's Chi-square test. In my setting, however, this test is inappropriate, because it requires a "large" number of observations *in each bin*, i.e.: for each j, it requires a large number of observations of the jth equivalence class of traces in the empirical observation $\hat{P}_i$. In particular, the often cited requirement of "at least five" observations each is not met for many programs. Also, the "number of bins", i.e.:

[6] in a fixed, but arbitrary ordering of these equivalence classes

the number $|\Theta/{\sim_L}|$ of different possible L-observations are usually high. Hence instead of a Chi-square test, I use a recent test for such *sparse* and *high-dimensional* multinomial distributions due to Plunkett and Park[PP18], which is based on (an unbiased estimator of) the Euclidean distance between P_i and P'_i.

For the rejection of the null hypothesis $H_0^{\geq\epsilon}$, I use a *goodness of fit* test due to Wellek ([Wel10], Section 9.1). The distance between two multinomial distributions $P_i, P_{i'}$ there is defined as the Eucledian distance

$$0 \leq \quad d(P_i, P_{i'}) = \sqrt{\sum_j |p_{i,j} - p_{i',j}|^2} \quad \leq 1$$

between the corresponding vectors. The null hypothesis hence formally is:

1. $H_0^{\geq\epsilon} \ : \ d(P_i, P_{i'}) \geq \epsilon$

Remark 16.4.2. Unlike Plunkett's and Park's test, Wellek's test is a *one* sample test. It is originally designed for the test of goodness-of-fit of distributions $P_i, P_{i'}$, based on one empirical distribution (say: $\hat{P}_{i'}$, obtained from sampling the a priori unknown distribution $P_{i'}$) and one *fully specified reference* distribution (say: P_i), i.e.: one a priori fully specified vector $(p_{i,1}, \ldots, p_{i,j}, \ldots)$.

Strictly speaking, what I need is a two-sample test, i.e.: a test for goodness-of-fit based on *two* empirical distributions. I shoehorn Wellek's test into my situation by employing Wellek's test *twice*, each time treating *one* of the two empirical distributions (say: $\hat{P}_i$) as the reference distribution (i.e: by assuming $\hat{P}_i = P_i$). I then reject the null hypothesis $H_0^{\geq\epsilon}$ only if both these two instances of Wellek's test do so.

Presumably, a *proper* two-sample goodness-of-fit test is possible by basing the test-statistics on the "two-sample"-variance estimator $\hat{\sigma}_k^2$ (from [PP18], Equation 9), instead of the "one-sample"-variance estimator v_n^2 (from [Wel10], Equation 9.8), but I did not pursue this.

Definition 16.4.1. The full ad hoc test for probabilistic noninterference for L-equivalent inputs i, i' to a given program works as follows:

1. Choose confidence level α (say: ≤ 0.05), maximal distance ϵ (say: < 0.01), and an initial value k (say: ≥ 10).
2. Sample L-observations of executions of G both for input i and for input i' to obtain a number $n = 2^k$ of total samples, each.
3. If both $H_0^{\geq\epsilon}$ is rejected by the test due to Wellek, *and* $H_0^{=}$ is rejected by the test due to Plunkett/Park, then report failure.
4. If $H_0^{=}$ is rejected by the test due to Plunkett/Park, then finish and report that G is *not probabilistically noninterferent*, with the pair (i, i') being a counterexample.
5. If $H_0^{\geq\epsilon}$ is rejected by the test due to Wellek, then finish and report that the pair (i, i') is *no counterexample* to the claim that G is probabilistically noninterferent.
6. Otherwise, increment k by one and continue at step 2.

In my automated experiments, I chose $\alpha = 0.0000001$, $\epsilon = 0.009$, and initialized $k = 12$ (such that initially, $n = 4096$).

The step 3 is not essential, and could have been left out. If triggered, it indicates evidence for a very small, but non-zero difference

$$0 < d(P_i, P_{i'}) < \epsilon$$

Step 3 could as well just report that G is *not* probabilistically noninterferent (as is done in step item 4). I inserted step 3 merely to determine frequency of situation described in Remark 16.4.1. In my automated experiments, I did indeed observe such a situation, but only *once* for 20000 randomly generated programs (and runs of the statistical test).

The doubling of sample size in each iteration due to step 6 is meant to reduce the probability of type I errors of either test, in lieu of proper *sequential hypothesis testing*.

By running this ad hoc test for two low-equivalent inputs i, i' and 20000 randomly generated For-programs *for which the* RLSOD *crite-*

rion claims probabilistic noninterference, I was able to validate Theorem 16.3.1.

Observation 16.4.1 (Empirical validation of Corollary 1 in [Bis+18b])**.** For at least 20000 randomly generated For-programs P (only counting those programs P such that the RLSOD-criterion claims probabilistic noninterference), manual inspection of all those program P for which the ad hoc test from Definition 16.4.1

- failed (via step item 3), or
- reported a counter-example (via step item 4)

revealed those P to be *in fact* probabilistically noninterferent[7].

For my choice of α, ϵ and initial number of samples $n = 2^k$ as above, the ultimately required number of samples is $n = 2^{12} = 4096$ for the majority of programs, but (rarely) goes up as high as $n = 2^{18} = 262144$. At the same time, false claims of counterexamples are rare. I observed just thirteen in total.

If I choose instead a confidence level of, e.g., $\alpha = 0.01$, then such errors become more common (on the order of roughly one per 100 checked programs).

In total I manually inspected 13 reported programs, all of which turned out to be probabilistically noninterferent, just as was claimed by the RLSOD-criterion in each case.

Efficiency of the Statistical Test

I designed the ad hoc empirical test to gain confidence in the correctness of the RLSOD-criterion from Section 16.3, before the correctness

[7] i.e.: manual inspection revealed those to be a type I error of the Plunkett/Park test, erroneously rejecting the null hypothesis $H_0^{=}$

proof from [Bis+18b] was available. I also use this test to check my new timing dependence based criterion coming up in Section 16.6.

But I can gain confidence in correctness by such a test only if I have confidence that the whole process of generating random For-programs and then testing the generated program on *one* pair (i, i') of L-equivalent inputs is indeed capable of exposing *faulty* criteria.

One could argue that what really needed to be done was to run the statistical tests for *all* pairs (i, i') of L-equivalent inputs, which of course is infeasible if not impossible[8]. But in practice, even using two fixed inputs i, i gives counterexample for faulty criteria relatively quickly. For example, when I tried to validate the *unsound* criterion obtained from RLSOD by omitting the rule CDA, the 232th randomly generated program exposed this unsoundness. This program — which is *not* probabilistically noninterferent, but accepted by this unsound criterion — is shown in Figure 16.3 (left). Here, the *order* of the two observable reads in line 8 and line 19 is influenced by the secret value read in line 2.

[8] This is different from the situation in Section 6.3, 6.6 and 9.5, in which *due to the simplicity* of the underlying notion of *input*, I *was* able to exhaustively test all equivalent pairs i, i' of inputs.

```
void main():
  z := read_h;
  y := read_l1;
  a := 0;
  fork thread_2();
  z := z;
  a := read_h;
  z := read_l1;
void thread_2():
  if (z*y <= 0)
    skip;
  else
    skip;
  for z {
    b := y * a;
  }
  y := read_h;
  fork thread_3();
  b := read_l2;
void thread_3():
  for 2 {
    print_l(1);
  }
  x := -z;
  x := y * z;
```

Figure 16.3: A randomly generated Example for the necessity of rule CDA

16.5 Imprecision of the RLSOD criterion

The example from Figure 16.2 on page 312 demonstrates how probabilistic noninterference is a less prohibitive notion of information flow security than low security observational determinism, which requires *any* executions for low-equivalent input to be observational indistinguishable (or in other words, it requires deterministic low observations).

Definition 16.5.1. Let i, i' be inputs; let $\Theta = T(i) \cup T(i')$. Low Security Observational Determinism holds iff

$$i \sim_{\mathrm{L}} i' \implies \forall t, t' \in \Theta.\ t \sim_{\mathrm{L}} t'$$

Since the example from Figure 16.2 is *not* low security observational deterministic, any sound static criterion for LSOD must reject it. On the other hand, this example *is* probabilistically noninterferent, and the RLSOD criterion (from Definition 16.3.3 on page 314) is indeed precise enough to accept it, hence for this instance, the RLSOD for probabilistic noninterference are an improvement over the previous LSOD criterion from [Gif12] and [GS15], which have to (and do) reject it.

In this section, I will first show that the RLSOD do not improve on the LSOD criterion on *every* program. I will then — by employing *timing dependence* $\rightarrow_{\mathrm{td}}$ — provide a criterion for probabilistic non-interference that improves on both the LSOD criterion and the RLSOD criterion. The new criterion will accept every program accepted by either of two previous others.

Giffhorns LSOD Criterion

First, I quickly review the LSOD criterion from [Gif12; GS15], in a style of presentation similar to that of the RLSOD criterion. Remember that the RLSOD criterion consisted of a rulesystem RLSOD of which the least solution cl is then submitted to a check RLSOD, and the program is deemed probabilistically noninterferent if cl admits the rule RLSOD.

In order to facilitate comparison with the RLSOD-criterion, I also present Giffhorns LSOD criterion in this form, but I then require *two* (simple) rule systems:

1. $\mathsf{LSOD}[\mathrm{cl}]$ whose least solution $\mathrm{cl}_{\mathsf{LSOD}} : N \to \mathcal{L}$ simply indicates the security level of nodes in the backward slice of each node.
2. $\mathsf{LSOD}[\mathrm{R}]$ whose least solution $\mathrm{R} \subseteq N$ is the set of nodes (potentially) influenced by a *data race*.

In order to present $\mathsf{LSOD}[\mathrm{R}]$, I need the notion of *interference-write* dependence.

Definition 16.5.2 ([Gif12], Definition 5.15)**.** For nodes n, m, interference-write-dependence via a variable x between n, m is defined as

$$\begin{array}{rrcl} & n \overset{x}{\leftrightarrow}_{\text{iw}} m & \iff & x \in \text{def}(n) \cap \text{def}(m) \text{ and } n\,\text{MHP}\,m \\ \text{and} & n \leftrightarrow_{\text{iw}} m & \iff & \exists x. n \overset{x}{\leftrightarrow}_{\text{iw}} m \end{array}$$

I also define interference-*read* dependence as follows[9]

$$\begin{array}{rrcl} & n \overset{x}{\to}_{\text{ir}} m & \iff & x \in \text{def}(n) \cap \text{use}(m) \text{ and } n\,\text{MHP}\,m \\ \text{and} & n \to_{\text{ir}} m & \iff & \exists x. n \overset{x}{\to}_{\text{ir}} m \end{array}$$

And ultimately, n, m are said to be in a data race if

$$n \leftrightarrow_{\text{race}} m \quad \iff \quad n \leftrightarrow_{\text{iw}} m \lor n \to_{\text{ir}} m$$

[9] This is just a repetition of Definition 16.3.1 from page 311 of interference dependence $\to_{\text{inter}}$, but only because that definition is based on the *may happen in parallel* notion, when in different settings it would have better been defined on a *may happen before* notion. Also see Remark 16.5.1

Remark 16.5.1. Unlike interference dependence $\xrightarrow{x}_{\text{inter}}$, interference *write* dependence $\xleftrightarrow{x}_{\text{iw}}$ *is* symmetric. If instead of a (symmetric) approximation MHP of *may happen in parallel* behavior, I had a (asymmetric) approximation MHB of *may concurrently happen before* behavior, I would need to demand $n \,\text{MHB}\, m \;\wedge\; m \,\text{MHB}\, n$ in the definition of $\xleftrightarrow{x}_{\text{iw}}$.

Similarly, I would then need to demand $n \,\text{MHB}\, m \;\wedge\; m \,\text{MHB}\, n$ in the definition of interference-*read* dependence $n \xrightarrow{x}_{\text{ir}} m$, while the definition of interference dependence $n \xrightarrow{x}_{\text{inter}} m$ would just demand $n \,\text{MHB}\, m$.

Definition 16.5.3 ([Bis+18b], Definition 12; following [GS15])**.** The rule system $\mathsf{LSOD}[\text{cl}]$ is:

$$\frac{m \in I}{\text{cl}_{\mathsf{LSOD}}(m) \sqsupseteq \text{ucl}(m)}\,\text{INPUT} \qquad \frac{n \to_{\text{cpdg}} m}{\text{cl}_{\mathsf{LSOD}}(m) \sqsupseteq \text{cl}_{\mathsf{LSOD}}(n)}\,\text{CPDG}$$

Definition 16.5.4 ([Bis+18b], Definition 12; following [GS15])**.** The rule system $\mathsf{LSOD}[\text{R}]$ is:

$$\frac{m_1 \leftrightarrow_{\text{race}} m_2}{m_2 \in \text{R}}\,\text{RACE} \qquad \frac{n \in \text{R} \qquad n \to_{\text{cpdg}} m}{m \in \text{R}}\,\text{CPDG}$$

Note that $\mathsf{LSOD}[\text{R}]$ is independent from the user classification ucl.

In the lattice $\mathcal{L}_2 = \{\text{L} \sqsubseteq \text{H}\}$, a program is then judged low (L) observationally deterministic by the LSOD-criterion if (given the least solutions $\text{cl}_{\mathsf{LSOD}}$ and R of these rule systems), the following three rules are admissible:

$$\frac{m \in O \qquad \text{ucl}(m) \sqsubseteq \text{L}}{\text{cl}_{\mathsf{LSOD}}(m) \sqsubseteq \text{L}}\,\mathsf{LSOD}_1 \qquad \frac{m \in O \qquad \text{ucl}(m) \sqsubseteq \text{L}}{m \notin \text{R}}\,\mathsf{LSOD}_2$$

$$\frac{m_1, m_2 \in I \cup O \qquad \text{ucl}(m_1), \text{ucl}(m_2) \sqsubseteq \text{L}}{\neg\, m_1 \,\text{MHP}\, m_2}\,\mathsf{LSOD}_3$$

Note that the propagation of *non-low input* along $\to_{\text{cpdg}}$ is completely separated (within $\text{cl}_{\mathsf{LSOD}}$) from the propagation of *races* along $\to_{\text{cpdg}}$ (within R). Also completely separated is the treatment of *order conflicts* in rule LSOD_3, which depends neither on $\text{cl}_{\mathsf{LSOD}}$ nor on R. Violations of rule LSOD_3 between two L-visible nodes, are always violations of low security observational determinism.

Contrasting these rules with the RLSOD rules, I note that

1. Rules INPUT and CPDG from rule system LSOD[cl] are the same as rules INPUT and CPDG from rule system RLSOD.
2. RLSOD has no rule corresponding to rule LSOD_3. This is because order conflicts prohibited by rule LSOD_3 are *only* violations of probabilistic non-interference if the probability of one of the two nodes (say: m_1) occuring before the other (m_2) is influenced by non-low input.
3. In LSOD[R], *any* race $m_1 \leftrightarrow_{\text{race}} m_2$ is propagated along $\to_{\text{cpdg}}$, no matter if the probability of m_1 occuring before m_2 in the $m_1 \leftrightarrow_{\text{race}} m_2$ was ever influenced by some non-low input.

   In contrast, in rule CDA from RLSOD, two nodes m_1, m_2 with undetermined execution order (as witnessed by the approximation $m_1 \, \text{MHP} \, m_2$) lead to propagation of a level $h \sqsupseteq \text{L}$ *only* if previously some node n[10] has been determined to be influenced by non-L input.

These considerations might lead one to belief that the RLSOD criterion is strictly “more precise” than the LSOD criterion, in the sense that whenever LSOD criterion accepts a given program, then also will the RLSOD criterion. But this is not so, in general. Consider the example on the left of Figure 16.4. It has only one possible low observation (a single output of 42), and hence is not only probabilistically noninterferent, but even low security observationally deterministic. The LSOD criterion is precise enough to detect this, since the $\to_{\text{cpdg}}$-

[10] subject to the *common dynamic ancestor* condition

backwardslice of the only L-output contains no high input (hence condition LSOD_1 is fulfilled), and the program contains no race (hence condition LSOD_2 is fulfilled). There is only one L-observable node n (at the print statement), hence there is no visible order-conflict, and condition LSOD_3 is fulfilled, too.

But the RLSOD criterion *rejects* this program. To see why, acknowledge that for the nodes corresponding to lines 4 and 5, $\mathrm{cl}(4), \mathrm{cl}(5) = \mathrm{H}$ by $\rightarrow_{\mathrm{cpdg}}$ (i.e.: by rule CPDG). Also, I have $\mathrm{cl}(7) \sqsupseteq \mathrm{cl}(4)$ due to rule CDA, because line 4 can happen after the execution of the common dynamic ancestor $c = 3 = \mathrm{icda}(7, 9)$, but before 7.

This particular kind of imprecision in the RLSOD criterion could be argued to be of not too much practical impact, for it requires a parallel thread *without* L observable output; had there been *any* L-observable output in thread 2, then the RLSOD criterion would have *rightfully* rejected the program, since then the delay due to loop in line 4 would have influenced the *relative execution time* of that output in thread 2, and the `print`$_l$`(42)` statement in line 7.

But also consider the example on the right of Figure 16.4, which is obtained from the example on the left by adding such a L-observable output to thread 2, and also replacing the loop in the main thread by a simple assignment. This example on the right is not low security observationally deterministic, but it *is* probabilistically noninterferent. Yet, the RLSOD criterion rejects it, by the same argument as for the example on the left.

In order to obtain a criterion for probabilistic noninterference more precise than the RLSOD-criterion, I propose to *separate* two kinds of influence of high input on low observations, which were *conflated* in the RLSOD-criterion:

1. The influence of high input on values at other nodes m, and on whether other nodes m are executed or not, as captured by the concurrent program dependence graph.

```
void main():
  h := read_h;
  fork thread_2();
  for h {
    skip;
  }
  print_l(42);
void thread_2():
  skip;
```

```
void main():
  h := read_h;
  fork thread_2();

  h2 := h

  print_l(42);
void thread_2():
  print_l(17);
```

Figure 16.4: Imprecision of the RLSOD-criterion

2. The influence of high input on the *relative execution order* of pairs (m_1, m_2) of nodes[11].

I will do this by use of *two* classifications $\mathrm{cl}_{⏱}$

1. A map $\mathrm{cl}_{⏱} : N \to \mathcal{L}$, from nodes n to security levels $\mathrm{cl}_{⏱}(n)$, *and*
2. a partial map $\mathrm{cl}_{⏱} : N \times N \hookrightarrow \mathcal{L}$, from pairs (m_1, m_2) of nodes to security levels $\mathrm{cl}_{⏱}(m_1,\ m_2)$

instead of just one classifications cl.

[11] which — lacking methods of *synchronization* — is possible in my program model only due to *delay* depending on high input

16.6 Timing Sensitivity for Probabilistic Noninterference

Consider the program from Figure 16.5. It is *not* probabilistically noninterferent, which can be argued as follows:

1. The value `h` of the secret input influences the length of the delay in line 8.
2. The length of the delay in line 8 influences the relative execution time of the two assignments to `delay2`, i.e.: it influences the probability of the assignment `delay2 := 0` occuring before the assignment `delay2 := 1000`, as opposed to vice versa.[12].
3. The read of variable `delay2` in line 11 sees the assignment `delay2 := 0` if that was executed after the assignment `delay2 := 1000`. Otherwise, it sees the assignment `delay2 := 1000`. But since the relative execution time of these two assignments was influenced by the value h (via the length of the delay in line 8), then so is the value of `delay2` in line 11.
4. The length of the delay in line 11 influences the relative execution time of the two publicly observable `print` statements.

Before I can present the rules that rigorously capture this kind of reasoning, I need to extend the notion of timing dependence from Chapter 10 to take into account spawn edges $\xrightarrow{\text{spawn}}$. The intuition here is that if for a spawn edge $m \xrightarrow{\text{spawn}} m'$, the node m is timing dependent on a node n in the control flow graph G, (i.e.: if $n \to^{G}_{\text{td}} m$), then also the timing of all nodes reachable from the entry node m' of the spawned thread depends on n.

[12] Line 8 also delays both the `print(42)` statement and the `print(17)` statement, but it delays them both *by the same amount*. Line 8 does not directly influence the *relative* execution time of these two `print` statements.

```
void main():
  delay1 = 0;
  h := read_h;
  if (h >= 0) {
    delay1 := 1000;
  }
  fork thread_2();
  for delay1 { skip; }
  delay2 := 0;
  fork thread_3();
  for delay2 { skip; }
  print_l(42);
```

```







void thread_2():
  delay2 := 1000;

void thread_3():
  print_l(17);
```

Figure 16.5: Interdependence of $\mathrm{cl}_{⏱}(m)$ and $\mathrm{cl}_{⏱}(m_1,\ m_2)$.

Definition 16.6.1. Given an (multi-threaded, labeled) control flow graph $G = (N, E)$ and spawn edges $\xrightarrow{\text{spawn}}$ connecting nodes from G, a node m is timing dependent on a node n in $G^{\text{spawn}} = \left(N, E \cup \xrightarrow{\text{spawn}}\right)$, and I write $n \to^{G^{\text{spawn}}}_{\text{timing}} m$ or just $n \to_{\text{timing}} m$ iff:

$$\begin{aligned} & n \to^{G}_{\text{td}} m \\ \text{or}\quad & n \to^{G}_{\text{td}} m' \xrightarrow{\text{spawn}} m'' \to^{*}_{G^{\text{spawn}}} m \text{ for some nodes } m', m'' \end{aligned}$$

Definition 16.6.2. The classification $\mathrm{cl}_{⏱}(\cdot) : N \to \mathcal{L}$ and the classification $\mathrm{cl}_{⏱}(\cdot,\ \cdot)\ N \times N \hookrightarrow \mathcal{L}$ are the least solution to the *mutually recursive* rule system TIMING consisting of rules

$$\frac{m \in I}{\mathrm{cl}_{⏱}\,(\sqsupseteq)\ \mathrm{ucl}\,(m)}\ \text{INPUT} \qquad \frac{n \to_{\text{cpdg}}\ m}{\mathrm{cl}_{⏱}(m) \sqsupseteq \mathrm{cl}_{⏱}(n)}\ \text{CPDG}$$

$$\frac{m_1 \overset{x}{\to}_{\text{inter}} m \qquad m_2 \overset{x}{\to}_{\text{inter}} m \qquad m_1\ \text{MHP}\ m_2}{\mathrm{cl}_{⏱}(m) \sqsupseteq \mathrm{cl}_{⏱}(m_1,\ m_2)}\ \text{RACE}_1$$

$$\frac{m_1 \overset{x}{\to}_{\text{inter}} m \qquad m_2 \overset{x}{\to}_{\text{data}}\ m \qquad m_1\ \text{MHP}\ m_2}{\mathrm{cl}_{⏱}(m) \sqsupseteq \mathrm{cl}_{⏱}(m_1,\ m_2)}\ \text{RACE}_2$$

and the rule

$$\frac{\begin{array}{c} m_1 \,\mathrm{MHP}\, m_2 \qquad c = \mathrm{icda}\,(m_1, m_2) \\ c \rightarrow^{*}_{G^{\mathrm{spawn}}} n \rightarrow^{*}_{G^{\mathrm{spawn}}} m_i \qquad n \rightarrow_{\mathrm{timing}} m_i \qquad i \in \{1,2\} \end{array}}{\mathrm{cl}_{⏱}\,(m_1,\ m_2) \sqsupseteq \mathrm{cl}_{⏱}\,(n)}\ \mathrm{CDA}$$

Here, $\mathrm{cl}_{⏱}\,(\cdot,\ \cdot)$ is defined on the set

$$\begin{array}{rl} & \{\, (m_1, m_2) \mid m_1 \,\mathrm{MHP}\, m_2 \ \wedge\ \mathrm{ucl}\, m_1 \sqsubseteq l \ \wedge\ \mathrm{ucl}\, m_2 \sqsubseteq l \,\} \\ \cup & \{\, (m_1, m_2) \mid \exists x.\ m_1 \overset{x}{\longleftrightarrow}_{\mathrm{iw}} m_2 \,\} \end{array}$$

of pairs (m_1, m_2) of nodes that are either both $l \in \mathcal{L}$ observable and may happen in parallel, or are interference-write dependent.

A program is then judged probabilistically noninterferent at level $l \in \mathcal{L}$ by the TIMING-criterion if (given the least solution $\mathrm{cl}_{⏱}$ of the rule system TIMING), the following two rules are admissible:

$$\frac{m \in O \qquad \mathrm{ucl}\,(m) \sqsubseteq l}{\mathrm{cl}_{⏱}\,(m) \sqsubseteq l}\ \mathsf{TIMING}_1$$

$$\frac{m_1 \,\mathrm{MHP}\, m_2 \qquad m_1, m_2 \in O \cup I \qquad \mathrm{ucl}\,(m_1), \mathrm{ucl}\,(m_2) \sqsubseteq l}{\mathrm{cl}_{⏱}\,(m_1,\ m_2) \sqsubseteq l \text{ and } \mathrm{cl}_{⏱}\,(m_1,\ m_2) \sqsubseteq l}\ \mathsf{TIMING}_2$$

Replaying the argument at the beginning of this chapter, in which I showed that the example from Figure 16.5 is *not* probabilistically noninterferent,

1. was an instance of rule CPDG,
2. was an instance of rule CDA,
3. was an instance of rule RACE_2 and
4. was an instance of rule CDA.

In the solution $\mathrm{cl}_{⏱}$ I have both $\mathrm{cl}_{⏱}\,(12) = \mathrm{L}$ and $\mathrm{cl}_{⏱}\,(24) = \mathrm{L}$ for the two L-observable `print` statements, indicating that the values printed by these statements do not depend on high input, and whether these statements are executed (and: how often) also does not depend on high input. Rule TIMING_1 *is* fulfilled. On the other hand, I *do* have $\mathrm{cl}_{\text{⏱}}(12, 14) = \mathrm{H}$, indicating that *the relative execution order* of these two print statements may depend on high input. The rule TIMING_2 is *not* fulfilled. The program is rejected by the TIMING timing criterion, which could *not* prove it to be probabilistically noninterferent.

I claim that the TIMING criterion is *sound*, i.e.: that any program accepted by the TIMING criterion (given some user classification ucl) is probabilistically noninterferent with regard observations at every security level $l \in \mathcal{L}$, as defined in Subsection 16.3.1. For the simple lattice $\mathcal{L} = \{\mathrm{L} \sqsubseteq \mathrm{H}\}$, I gathered empirical evidence in support of this claim by use of the statistical test for probabilistic noninterference (Definition 16.4.1 from 320).

Observation 16.6.1 (Soundness of the TIMING criterion, empirical)**.** For at least 20000 randomly generated For-programs P (only counting those programs P such that the TIMING-criterion claims probabilistic noninterference), manual inspection of all those program P for which the ad hoc test from Definition 16.4.1

- failed (via step item 3), or
- reported a counter-example (via step item 4)

revealed those P to be *in fact* probabilistically noninterferent

Since the statistical test may take considerable time (roughly between 5 and 100 seconds for the randomly generated programs), I used in Observation 16.6.1 the same randomly generated programs as before in the empirical validation of the RLSOD criterion (Observation 16.4.1). Those programs that required manual inspection all also passed the RLSOD criterion, so these were the same 13 programs as before.

The two programs from Figure 16.4 are probabilistically noninterferent and accepted by the TIMING criterion, but not by the RLSOD criterion. In general, the TIMING-criterion is more precise than the RLSOD criterion.

Observation 16.6.2 (Precision of the TIMING criterion, relative to the RLSOD criterion)**.** Let cl be the least solution to the rule system RLSOD (Definition 16.3.3) and $\mathrm{cl}_{\text{⏱}}$ be the least solution to the rule system TIMING (Definition 16.6.2).

Then

$$\begin{array}{rlcl} & \mathrm{cl}_{\text{⏱}}\,(m) & \sqsubseteq & \mathrm{cl}\,(m) \\ \text{and} & \mathrm{cl}_{\text{⏱}}\,(m_1,\ m_2) & \sqsubseteq & \mathrm{cl}\,(m_1) \sqcup \mathrm{cl}\,(m_2) \end{array}$$

for all nodes m, and all pairs (m_1, m_2) in the domain of $\mathrm{cl}_{\text{⏱}}$. Consequently in the lattice $\mathcal{L}_2 = \{\mathrm{L} \sqsubseteq \mathrm{H}\}$[13], if the RLSOD criterion holds at level l, then also does the TIMING criterion.

Remark 16.6.1. Specifically, the two inequations in Observation 16.6.2 hold for the "slicing" subset-lattice $\mathcal{L} = (2^N, \subseteq)$, when initializing each node n with $\mathrm{ucl}\,(n) = \{n\}$.

[13] and more generally: all "linear" lattices, in which $l_1 \sqcup l_2 \in \{l_1,\ l_2\}$

Summary

- For concurrent programs with probabilistic schedulers, probabilistic non-interference the appropriate security property.
- In line with previous research, the execution time is considered *externally* unobservable.
- Still, the relative execution time of memory accesses in concurrent threads makes internal differences in timing externally observable.
- The LSOD criterion, the RLSOD criterion, and the new TIMING criterion guarantee probabilistic non-interference.
- I substantiate this claim by extensive random testing.
- In these tests, probabilistic non-interference is validated by a statistical test.
- The LSOD criterion is very strict. The RLSOD criterion more liberal in some, but not all cases.
- The new TIMING criterion, based on timing dependence $\rightarrow_{\text{td}}$, is more liberal than both the LSOD criterion and the RLSOD criterion.

17 Timing Sensitivity with JOANA

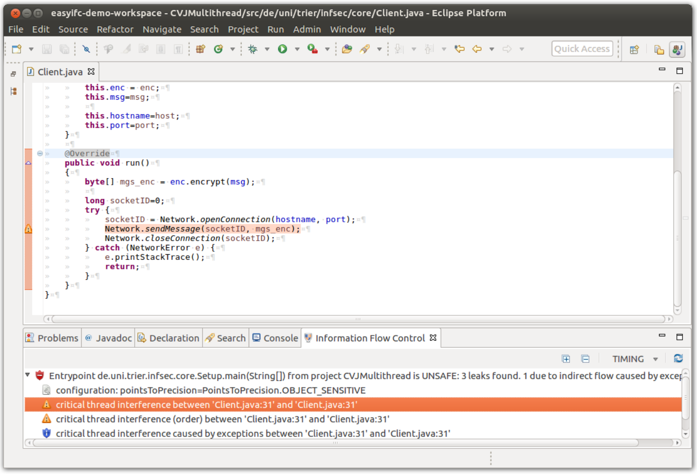
"Our product is still totally DeepArcher?"
"Which is ..."
"Like 'departure', only you pronounce it DeepArcher?"
"Zen thing," Maxine guesses.
"Weed thing."

(Thomas Pynchon — Bleeding Edge)

In Chapter 16, I introduced the new TIMING criterion for concurrent programs. I presented it for control flow graphs in a simple imperative language. But the TIMING criterion can be also be used for the analysis of concurrent programs more complex languages. In fact, I applied the TIMING criterion to a simple case study in the Java programming language, using an implementation in the JOANA[1] system. In this section I show that here, too, the TIMING criterion can improve on the RLSOD criterion.

The JOANA system[HS09; Sne+14] implements information flow analyses for Java programs. For the analysis of sequential programs, JOANA provides an (interprocedural) program dependence graph ("system" dependence graph), consisting of control and data dependencies as well summary dependencies which allow calling-context sensitive slicing. Such slices can be used for the verification of (sequential) non-interference[Was10]. The JOANA program dependence

[1] As explained by Jürgen Graf: "JOANA means **J**ava **O**bject-sensitive **ANA**lysis in case you wondered :)"


easyifc-demo-workspace - CVJMultithread/src/de/uni/trier/infsec/core/Client.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Admin Window Help
Quick Access
Client.java
Problems
Javadoc
Declaration
Search
Console
Information Flow Control
TIMING

```
        this.enc = enc;
        this.msg=msg;

        this.hostname=host;
        this.port=port;
    }

    @Override
    public void run()
    {
        byte[] mgs_enc = enc.encrypt(msg);

        long socketID=0;
        try {
            socketID = Network.openConnection(hostname, port);
            Network.sendMessage(socketID, mgs_enc);
            Network.closeConnection(socketID);
        } catch (NetworkError e) {
            e.printStackTrace();
            return;
        }
    }
}
```

```
Entrypoint de.uni.trier.infsec.core.Setup.main(String[]) from project CVJMultithread is UNSAFE: 3 leaks found. 1 due to indirect flow caused by excep
    configuration: pointsToPrecision=PointsToPrecision.OBJECT_SENSITIVE
    critical thread interference between 'Client.java:31' and 'Client.java:31'
    critical thread interference (order) between 'Client.java:31' and 'Client.java:31'
    critical thread interference caused by exceptions between 'Client.java:31' and 'Client.java:31'
```

Figure 17.1: The JOANA System

graph for Java programs are described in detail in [Ham09; Gra16]. Slicing Algorithm are described in [Ham09; Gif12]. For concurrent programs, the JOANA program dependence graph is extended with interference dependencies to form the concurrent program dependence graph, as described in the same two theses. Also described there are algorithms for slicing of concurrent programs. The implementation of the LSOD criterion and the computation of precise may-happen-in-parallel information in JOANA is described in [Gif12].

Martin Mohr, Simon Bischof and I implemented the RLSOD criterion and my TIMING criterion based on JOANA dependency graphs. Immediate common ancestors icda (m_1, m_2) and timing dependence $n \to_{\text{timing}} m$ are implemented as described in Chapter 16 (specifically: they are calling-context insensitive). On the other hand, dependencies due to the concurrent program dependence graph $n \to_{\text{cpdg}} m$ are computed calling-context sensitively, i.e.: rule CPDG is replaced by an iterated two phase slice (see, e.g., [Gif12]).

17.1 Precision of the TIMING criterion for Java

In a case study in [Bis+18b], we applied the JOANA implementation of the RLSOD criterion to a concurrent Java implementation of a small client/server application. We used the framework for the cryptographic verification of Java Programs due to [KTG12; Küs+14]. In this framework, the concrete implementation of cryptographic primitives (e.g., encryption and decryption) are replaced by *idealized* implementations, in which the content of encrypted messages does *not* depend on the content of plain-text messages. Non-interference of the idealized implementation, together with conventional cryptographic assumptions, then imply computational indistinguishability for the system with a *real* implementation of cryptographic primitives.

In the case study from [Bis+18b], the goal was to prove that a client's choice between two possible plain text messages remains secret to an attacker who observes the network communication between the clients and the server. This is a simplification of a verification goal in e-voting systems. There, the clients choose between two or more alternative candidates, but only the server is allowed to learn the vote (by decryption of the message). The client's choice is modeled as the input to a variable `secret_bit`, resulting in a corresponding user classification $\mathrm{ucl}\,(\cdot) = \mathrm{H}$. In the JOANA system, this is achieved by annotating the corresponding Java code with a `@Source` annotation. We also annotated the *private key* used for decryption of messages with such an annotation. The attackers observation capabilities are modeled by a `@Sink` annotation at the method `Network.sendMessage(...)`.

Using the classification cl for the RLSOD criterion inferred by JOANA, and then additional manual inspection, we concluded that any possible information leak must be due to the execution time of the `Encryptor.encrypt()` method. But we could not conclude from RLSOD criterion the that the execution time must be independent from the variable `secret_bit`.

On the other hand, the JOANA implementation of the TIMING criterion for the same program reports *no* violation of probabilistic noninterference due to the variable `secret_bit`.[2] The critical code is

```
for(int i=0; i<msg1.length; ++i) {
    msg[i] = (secret_bit ? msg1[i] : msg2[i]);
}
```

in which the plain text message is chosen according to `secret_bit`. The RLSOD criterion must compute $\mathrm{ucl}\,(n) = \mathrm{H}$ for the corresponding control flow graph nodes n in the loop body, and then by rule CDA also conclude $\mathrm{ucl}\,(m) = \mathrm{H}$ also for the node m corresponding to network communication at `Network.sendMessage(...)`. But the TIMING criterion does *not* propagate the classification $\mathrm{cl}_{⏱}\,(n) = \mathrm{H}$ to the relative timing of any two nodes $(m_1,\ m_2)$. To see this, consider the CDA rule for TIMING. There, nodes m_i are *not* timing dependent on any node n in the loop body. Specifically, they are not timing dependent on the choice node implicit in the expression

```
secret_bit ? msg1[i] : msg2[i]
```

[2] Although just like the RLSOD criterion, it *does* report a violation due to the private encryption key.

17.2 Scalability of the TIMING criterion for Java

In the case study from the previous section, the program consisted of 550 lines of code in 16 classes. Since the JOANA analyses are *whole program* analyses, the code was analyzed together with used components of the Java standard libraries. The total analysis used 2713 megabytes RAM and finished within 2.9 seconds. This includes the computation of the JOANA concurrent program dependence graph, may happen in parallel information, and the computation of the TIMING criterion. The latter took a total of 297 milliseconds, of which the computation of timing dependencies $n \rightarrow_{\text{timing}} m$ required 108 milliseconds, and the computation of the control flow graph “chops” implicit in the TIMING classification rule CDA took 97 milliseconds.

In order to give an idea of the scalability of the TIMING criterion, I also applied it to the *Apache FtpServer*[Fou19]. In addition to the Java standard libraries, the analyzed core of this program makes use of 125818 lines of library source code, and consists of 20645 lines of code[3]. The total analysis used 87 gigabytes RAM and finished within 10996 seconds ($\approx$ 3h03m). The computation of the TIMING criterion took a total of 9844 seconds ($\approx$ 2h44m), of which the computation of timing dependencies $n \rightarrow_{\text{timing}} m$ required 4306 seconds ($\approx$ 1h11m), while the computation of the control flow graph “chops” implicit in the TIMING classification rule CDA took 5343 seconds ($\approx$ 1h29m). Within the remaining time (1152 seconds), the computation of the concurrent program dependence graph (including summary edges) graph took 640 seconds, and the may-happen-in-parallel information required 197 seconds.

The *Privacy Crash Cam* is a system developed at the KASTEL Competence Center for Applied Security Technology. I ran the TIMING criterion on its web-service component, which makes available crash-incident related videos from car *dash-cams* to various stakeholders (drivers, other parties involved in an car-accident, law-enforcement).

[3] all as counted by the `cloc` utility[Dan18], ignoring comments

	total	TIMING	$\rightarrow_{\text{timing}}$	"chops"	other	$\rightarrow_{\text{cpdg}}$	MHP
apache	10 996s	9 844s	4 306s	5 343s	1 152s	640s	197s
pcc	121 549s	103 304s	25 278s	77 066s	18 245s	9 112s	3 870s

Figure 17.2: Run Time of TIMING Based Analysis for Concurrent Java in JOANA

In addition to the Java standard libraries, this web service component makes use of 495607 lines of library source code, and consists of 2894 lines of code. The analysis used 305 gigabytes RAM. See Figure 17.2 for times required for the analysis.

All times in this chapter were measured on a high end "computation server" class PC with an Intel Xeon Gold 6230 CPU at 2.10 GHz base frequency with 512GB RAM.

Summary

- An implementation of the new TIMING criterion for concurrent Java is available in the JOANA system.
- In a simple case study, it improves in precision on the RLSOD criterion of the JOANA system.
- The computation of the TIMING criterion is feasible for a concurrent server application of ≈ 100 000 lines of code.

18 Summary and Future Work

Arrakis teaches the attitude of the knife — chopping off what's incomplete and saying: "Now, it's complete because it's ended here."

(Frank Herbert — Dune)

In this thesis, I introduced timing sensitive control dependence $\rightarrow_{\text{tscd}}$ as a natural modification to nontermination sensitive control dependence $\rightarrow_{\text{ntscd}}$. For control flow graphs with unique exit node (also: for all reducible graphs), I can compute $\rightarrow_{\text{tscd}}$ using the generalized control dependence Algorithm 1. For such graphs, timing sensitive postdominance $\sqsupseteq_{\text{TIME[FIRST]}}$ is transitive, and I can computed it by a modification of an algorithm for nontermination sensitive postdominance $\sqsupseteq_{\text{MAX}}$. I used timing sensitive control dependence to support a static information flow analysis for concurrent programs, and an information flow analysis sensitive to timing channels due to micro-architectural effects. For the first analysis, I provided a practical implementation for Java programs in the JOANA system. For the second, I provided a prototype implementation for programs with variables and arrays.

No work is ever truly finished, and mine is no exception. In the following, I want to mention some possible avenues for future work on further improvements of my results.

Calling-Context Sensitivity In the JOANA analysis for concurrent Java programs, neither timing dependence $\rightarrow_{\text{timing}}$, nor the classifications $\text{cl}_{⏱}$ are computed calling-context sensitively. I suspect that calling-context timing dependence could be achieved, possibly by enriching control flow graphs with timing sensitive "control flow summary edges": Whenever in a called procedure, timing sensitive postdominance is established between exit and entry node (for some k), equip corresponding call-sites with a "summary" edge with timing cost k. Otherwise, equip call sites with a summary edge with timing cost $\top$, to be handled appropriately.

Weakening of Timing Sensitivity Timing sensitive control dependence $n \rightarrow_{\text{tscd}} m$ holds unless all successor of a branch node n reach node m after exactly k units of time. Is it possible (and useful?) to consider not only exact matches of time, but also approximate matches? Possibly, by computing intervals $[k_{min}, k_{max}]$ of timing costs instead of just one value k?

"Declassification" of Timing Leaks Pragmatically, a user of timing sensitive analysis will want to ignore timing in selected parts of the program. Perhaps this is possible by allowing him to axiomatically introduce additional edges in the transitive reduction $<_{\text{TIME[FIRST]}}$ of timing sensitive postdominance? Note that if this is done at nodes n with no current successor in $<_{\text{TIME[FIRST]}}$, it remains a pseudo-tree. Such *timing declassifications* could also be provided by an auxiliary more costly but more precise sound analysis of the relevant program part.

Micro-Architectural / Cache Dependencies for *real* Architectures In order to be practically useful, (approximate) cache dependencies must be computed for the actual binaries run on actual hardware, using a "usefully realistic" cache model of that hardware. I expect my analysis to work practically unchanged for common instruction set architectures. But it will only be useful together with a precise analysis of memory accesses, i.e.: for each machine operation l, a "as small as possible" static approximation of those memory blocks that may be accessed by l, which in turn will require a precise analysis of values in, e.g., registers used as offsets in relative address accesses.

New Algorithm for Micro-Architectural dependencies Micro-Architectural dependencies first compute the full (sometimes huge) graph G_α, starting from some initial cache state. Is it possible instead to compute, for each node m, only a relevant part of G_α by going *backwards* from m?

Approximate Cache Dependencies I introduced approximate cache dependencies only for LRU caches. But I expect similar notions to be possible also for, e.g., *pseudo* LRU caches.

A More Efficient TIMING **Criterion** As stated, the rule CDA in the TIMING Criterion require the computation of numerous "control flow chops". I strongly suspect that the explicit computation of these chops can be avoided by use of an appropriate data structure.

Appendices

A Proofs

Very deep. You should send that in to the Reader's Digest. They've got a page for people like you.

(Douglas Adams — The Hitchhiker's Guide to the Galaxy)

A.1 Nontermination (In-)Sensitive Control Dependence in Arbitrary Graphs

Proof of Lemma 3.2.1 on page 20: If $x \neq y$, then $x \sqsupset y \sqsupseteq y$. Also, let $x' \sqsupset \text{␣} \sqsupseteq y$. Then $x' \sqsupseteq x$ by transitivity.

For $\text{EQ}_2^{\sqsupseteq}$, I only need to show $\text{ipdom}_{\sqsupseteq}(x) \subseteq \text{ipdom}_{\sqsupseteq}(y)$. Now, for any $x' \in \text{ipdom}_{\sqsupseteq}(x)$, I have $x' \sqsupset \text{␣} \sqsupseteq x \sqsupseteq y$. Also, for any y' such that y' 1-$\sqsupseteq y$, I have $y' \sqsupset \text{␣} \sqsupseteq y \sqsupseteq x$. Because $x' \in \text{ipdom}_{\sqsupseteq}(x)$, I conclude $y' \sqsupseteq x'$, and hence $x' \in \text{ipdom}_{\sqsupseteq}(y)$.

For $\text{EQ}_3^{\sqsupseteq}$, assume $x \neq y$. Then $y \sqsupset x \sqsupseteq z$. Also, if $y' \sqsupset \text{␣} \sqsupseteq z$, then $y' \sqsupseteq x \sqsupseteq y$ because $x \in \text{ipdom}_{\sqsupseteq}(z)$. □

Proof of Lemma 3.2.2 on page 22:

1. $\text{PDF}^{\text{local}}_{\sqsupseteq}(x) \subseteq \text{PDF}_{\sqsupseteq}(x)$ because $x \sqsupseteq x$.
2. $\text{PDF}^{\text{up}}_{\sqsupseteq}(z) \subseteq \text{PDF}_{\sqsupseteq}(x)$ for $x \in \text{ipdom}_{\sqsupseteq}(z)$:
   Let $y \in \text{PDF}^{\text{up}}_{\sqsupseteq}(z)$. I have $\neg\ x$ 1-$\sqsupseteq y$ by definition. Also, from $y \in \text{PDF}_{\sqsupseteq}(z)$ I obtain some s such that $y \rightarrow_G s$ and $z \sqsupseteq s$. From x 1-$\sqsupseteq z$ and due to reflexivity I am done, because $x \sqsupseteq z \sqsupseteq s$.
3. I show for any x, s such that $(x,s) \in \text{ipdom}^*_{\sqsupseteq}$ that the rule

$$\frac{\neg\ x \text{ 1-}\sqsupseteq y \qquad y \rightarrow_G s \qquad x \sqsupseteq s}{y \in \text{PDF}^{\text{local}}_{\sqsupseteq}(x) \ \vee\ y \in \bigcup_{\{z \,\mid\, x \in \text{ipdom}_{\sqsupseteq}(z)\}} \text{PDF}^{\text{up}}_{\sqsupseteq}(z)}$$

   is admissible, by induction on $(x,s) \in \text{ipdom}^*_{\sqsupseteq}$.

   I $x = s$.

   Let y be as in the rule's premise. Then $y \in \text{PDF}^{\text{local}}_{\sqsupseteq}(x)$ by definition.

II $x \in \mathrm{ipdom}_{\sqsupseteq}(z)$ and $(z, s) \in \mathrm{ipdom}^{*}_{\sqsupseteq}$ for some $z \in N$.
Let y be as in the rule's premise, i.e.:

$\neg\ x\ 1\text{-}\sqsupseteq y \quad y \rightarrow_G s \quad x \sqsupseteq s$

In order to exploit the induction hypothesis, I want to show:

$\neg\ z\ 1\text{-}\sqsupseteq y \qquad\qquad z \sqsupseteq s$

In order to obtain a contradiction, assume $z\ 1\text{-}\sqsupseteq y$. From $x \in \mathrm{ipdom}_{\sqsupseteq}(z)$, I also have $x\ 1\text{-}\sqsupseteq z$, i.e.:

$$x \sqsupset \text{␣} \sqsupseteq z \sqsupset \text{␣} \sqsupseteq y$$

which contradicts $\neg\ x\ 1\text{-}\sqsupseteq y$, due to the transitivity of $\sqsupseteq$.

Also, $z \sqsupseteq s$ because $\mathrm{ipdom}^{*}_{\sqsupseteq} = \sqsupseteq$.

Now, from the induction hypothesis, either:

i. I have $y \in \mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(z)$,

in which case I show $y \in \mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z)$ as follows:

First, I conclude $y \in \mathrm{PDF}_{\sqsupseteq}(z)$ as before.

Also, let $x' \in \mathrm{ipdom}_{\sqsupseteq}(z)$. Then $x \sqsupseteq x'$ and $x' \sqsupseteq x'$. Assume $x \neq x'$, because otherwise immediately $\neg\ x'\ 1\text{-}\sqsupseteq y$. If I had $x'\ 1\text{-}\sqsupseteq y$, then:

$$x \sqsupset x' \sqsupset \text{␣} \sqsupseteq y, \text{ i.e., via reflexivity, } x \sqsupset x' \sqsupseteq y,$$

in contradiction with the choice of y.

or

ii. I obtain a node z' such that $z \in \mathrm{ipdom}_{\sqsupseteq}(z')$ and $y \in \mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z')$

In this case, I need to show: $y \in \mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z)$.

- By definition of $\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z')$, and because $z \in \mathrm{ipdom}_{\sqsupseteq}(z')$, I have $\neg\, z \text{ 1-}\!\sqsupseteq y$.
- From $y \in \mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(z')$ I have $y \in \mathrm{PDF}_{\sqsupseteq}(z')$ by definition. Hence: $z' \sqsupseteq s'$ for some s' such that $y \rightarrow_G s'$. Because $z \in \mathrm{ipdom}_{\sqsupseteq}(z')$, this means that

  $$z \sqsupset \textvisiblespace \sqsupseteq z' \sqsupseteq s'$$

  This shows $z \sqsupseteq s'$ due to transitivity, and hence: $y \in \mathrm{PDF}_{\sqsupseteq}(z)$.
- It remains to show for arbitrary $x' \in \mathrm{ipdom}_{\sqsupseteq}(z)$: $\neg\, x' \text{ 1-}\!\sqsupseteq y$. But this follows just as it did in case i.

□

Proof of Lemma 3.2.3 on page 23: Given $y \rightarrow_G x$, I show

$$x \text{ 1-}\!\sqsupseteq y \iff x \in \mathrm{ipdom}_{\sqsupseteq}(y)$$

Assume $x \text{ 1-}\!\sqsupseteq y$, and let let x' be any node such that $x' \text{ 1-}\!\sqsupseteq y$ If $x' \neq y$, then $x' \sqsupseteq x$ by $x' \sqsupseteq y$ (transitivity) and rule $\mathrm{CL}^{\rightarrow_G}$. If $x' = y$, let v be some node such that $y \sqsupset v \sqsupseteq y$. Then, by rule $\mathrm{CL}^{\rightarrow_G}$ I have $v \sqsupseteq x$, and hence $x' = y \sqsupseteq x$, as well, which proofs $x \in \mathrm{ipdom}_{\sqsupseteq}(y)$.

The reverse implication follows directly from the definitions. □

Proof of Lemma 3.2.4 on page 25: Given $y \in \mathrm{PDF}_{\sqsupseteq}(z)$, I show

$$x \in \mathrm{ipdom}_{\sqsupseteq}(y) \quad \iff \quad \exists x' \in \mathrm{ipdom}_{\sqsupseteq}(z)\,.\, x' \text{ 1-}\!\sqsupseteq y$$

The implication $\implies$ is trivial, given the assumption on x.

For the reverse implication $\Longleftarrow$ let x' as provided. Because both x, x' are in $\in$ $\text{ipdom}_{\sqsupseteq}(z)$, I obtain $x \sqsupseteq x'$ and then — regardles whether $x = x'$ or not —

$$x \; 1\text{-}\!\sqsupseteq y$$

From this, if also $y \sqsupseteq x$, I immediately conclude $x \in \text{ipdom}_{\sqsupseteq}(y)$ from $\text{EQ}_1^{\sqsupseteq}$. Otherwise, because $\text{ipdom}_{\sqsupseteq}^{*} = \sqsupseteq$, I obtain via Observation 3.2.1 nodes w, v such that

$$w \in \text{ipdom}_{\sqsupseteq}(v) \;\text{ and }\; v \sqsupseteq y \;\text{ and }\; x \sqsupseteq w \sqsupseteq x \;\text{ and }\; \neg\, v \sqsupseteq x$$

but also, because of $\text{EQ}_3^{\sqsupseteq}$

$$x \in \text{ipdom}_{\sqsupseteq}(v)$$

If $z = v$, I conclude $z = y$, because otherwise: $z \sqsupset y \sqsupseteq y$, i.e.: $z \; 1\text{-}\!\sqsupseteq y$, in contradiction to $y \in \text{PDF}_{\sqsupseteq}(z)$. This shows $x \in \text{ipdom}_{\sqsupseteq}(y)$. This is also true if $v = y$.

This means I still have to show $x \in \text{ipdom}_{\sqsupseteq}(y)$ if $z \neq v$ and $y \neq v$. From $y \in \text{PDF}_{\sqsupseteq}(z)$, I obtain a node s such that $y \rightarrow_G s$ and $z \sqsupseteq s$. From $\text{CL}^{\rightarrow_G}$ and $v \sqsupseteq y$ I conclude $v \sqsupseteq s$. Now, I can use NoJoin to infer that either $z \in \text{ipdom}_{\sqsupseteq}(v)$ or $v \in \text{ipdom}_{\sqsupseteq}(z)$. But if $z \in \text{ipdom}_{\sqsupseteq}(v)$, then

$$z \sqsupset v \sqsupseteq y, \;\text{ i.e.: }\; z \; 1\text{-}\!\sqsupseteq y$$

in contradiction to $y \in \text{PDF}_{\sqsupseteq}(z)$. If, on the other hand, $v \in \text{ipdom}_{\sqsupseteq}(z)$, specifically: $v \; 1\text{-}\!\sqsupseteq z$, but then $v \sqsupseteq x$ because $x \in \text{ipdom}_{\sqsupseteq}(z)$, in contradiction to the choice of v.

$\square$

A.2 Postdominator Pseudoforests

Proof of Theorem 5.1.1 on page 41: First, i show $\sqsupseteq_{\mathrm{POST}} \subseteq \nu\mathsf{P}$. By the co-induction proof principle, I have to show

$$\sqsupseteq_{\mathrm{POST}} \quad \subseteq \quad \mathsf{P}\left(\sqsupseteq_{\mathrm{POST}}\right)$$

So let $m \sqsupseteq_{\mathrm{POST}} p$. The case $p = m$ is trivial, so assume $p \neq m$. I know that $\forall p \to_G x.\ x \sqsupseteq_{\mathrm{POST}} m$ by definition of $m \sqsupseteq_{\mathrm{POST}} p$, so: $(m, p) \in \mathsf{P}\left(\sqsupseteq_{\mathrm{POST}}\right)$.

Now, let me show $\sqsupseteq_{\mathrm{POST}} \supseteq \nu\mathsf{P}$, by assuming $\neg m \sqsupseteq_{\mathrm{POST}} p$, and showing $(m, p) \notin \nu\mathsf{P}$.
There must be some path $p \to_G^{\pi} n_x$ to the unique exit node n_x such that $m \notin \pi$. Specifically, $m \neq p$, so I cannot use rule $\mathsf{P}^{\mathrm{self}}$ to validate $(m, p) \in \nu\mathsf{P}$. Let x be the successor of p in π, so that $\pi = p, x, \pi'$. Then I can only validate $(m, p) \in \nu\mathsf{P}$ if I can validate $(m, x) \in \nu\mathsf{P}$. But I now have a *shorter* path $x \to_G^{x,\pi'} n_x$ such that $m \notin x, \pi'$, so by iterating I eventually find that I cannot validate $(m, p) \in \nu\mathsf{P}$ at all. □

Proof (Sketch) of Theorem 5.1.2 on page 42: The proof for $\sqsupseteq_{\mathrm{SINK}}$ is similar to the proof of Theorem 5.1.1. For $\mu\mathsf{D} \subseteq \sqsupseteq_{\mathrm{MAX}}$, by the induction proof principle I have to show:

$$\mathsf{D}\left(\sqsupseteq_{\mathrm{MAX}}\right) \quad \subseteq \quad \sqsupseteq_{\mathrm{MAX}}$$

i.e. I have to show:

$$\text{whenever } \forall p \to_G x.\ m \sqsupseteq_{\mathrm{MAX}} x \text{ and } p \to_G^{*} m \text{ then also } m \sqsupseteq_{\mathrm{MAX}} p$$

for $m \neq x$, but this follows directly from the definition. For $\sqsupseteq_{\mathrm{MAX}} \subseteq \mu\mathsf{D}$, I assume $m \sqsupseteq_{\mathrm{MAX}} n$. Let $m \neq n$, because otherwise I have $(m, n) \in \mu\mathsf{D}$ by rule $\mathsf{D}^{\mathrm{self}}$. Because of $m \sqsupseteq_{\mathrm{MAX}} n$, all paths from n to m in which m occurs only once (at the end) are cycle-free. Let π be such a path with maximal length among all such paths. Also denote by π_x for each successor x of n a path with maximal length among all such paths from x to m. Then all π_x are certainly shorter than π, and inductively I can assume $m \sqsupseteq_{\text{MAX}} x$ for all such x. But then $(m, n) \in \mu\text{D}$ by rule D^{suc}. □

Proof of Lemma 5.2.2 on page 45: If both $v \sqsupseteq z$ and $z \sqsupseteq v$ i'm done by rule $\text{EQ}_1^{\sqsupseteq}$. So by symmetry, it is enough to show

$$\neg z \sqsupseteq v \qquad \Longrightarrow \; v \in \text{ipdom}_{\sqsupseteq_{\text{MAX}}}(z)$$

So let $>_{\text{MAX}}$ be any transitive reduction of $\sqsupseteq_{\text{MAX}}$, and assume $\neg z \sqsupseteq v$ (i.e., even: $\neg z \sqsupset v$). From $v \sqsupseteq_{\text{MAX}} s$ and because $<_{\text{MAX}}$ is a pseudo forest, i know that there is exactly one sequence

$$v >_{\text{MAX}} \ldots >_{\text{MAX}} s$$

such that v appears exactly once. But since $\neg z \sqsupset v$ and $z \sqsupseteq_{\text{MAX}} s$, i know that z must appear in between, i.e. (since $v \neq z$) i have:

$$v >_{\text{MAX}}^{*} v' >_{\text{MAX}} z >_{\text{MAX}}^{*} s$$

I also have — because $x \in \text{ipdom}_{\sqsupseteq_{\text{MAX}}}(z)$ — some x' s.t.

$$x' >_{\text{MAX}}^{*} x >_{\text{MAX}}^{*} x' >_{\text{MAX}} z$$

But since $<_{\text{MAX}}$ is a pseudo forest, i have $x' = v'$, and also: v must lay on the cycle $x' >_{\text{MAX}}^{*} x >_{\text{MAX}}^{*} x'$, i.e.:

$$v' >_{\text{MAX}}^{*} v >_{\text{MAX}}^{*} v' >_{\text{MAX}} z$$

□

A.2.1 Proof of Correctness of the Workset implementation of Algorithm 5

On order to provide — apart from the empirical evidence – proof of correctness, I need some terminology. I want to express that any node

n for which I have not already determined its successor (if any) in $<_{\text{MAX}}$, either is still in the workset, or will be put there eventually.

Definition A.2.1. Let workset $\subseteq \text{COND}_G$ and $<$ be some pseudo-forest. Then the set of nodes *accessible* from workset is defined as the least fixed point of the rule system

$$\frac{n \in \mathsf{workset}}{n \in \mathsf{workset}^*}$$

$$\frac{w \in \mathsf{workset}^* \qquad n \in \text{COND}_G \qquad n \to_G y \qquad y <^* w}{n \in \mathsf{workset}^*}$$

Also, i define

$$<_{\mathsf{workset}} \quad = \quad < \ \cup \ \left\{ (w, m) \;\middle|\; w \in \mathsf{workset}^*,\ m \in \text{ipdom}_{\sqsupseteq_{\text{MAX}}}(w) \right\}$$

and write $<^*_{\mathsf{workset}}$ for $(<_{\mathsf{workset}})^*$.

Proof (Sketch): The algorithm establishes and then upholds the following invariants for $<$ (represented by IMDOM) and workset:

$$m \sqsupseteq_{\text{MAX}} n \quad \iff \quad n <^*_{\mathsf{workset}} m$$

$$n < m \quad \Longrightarrow \quad m \in \text{ipdom}_{\sqsupseteq_{\text{MAX}}}(n)$$

$$\mathsf{IMDOM}[n] = \bot \ \wedge \ \exists m \neq n. m \sqsupseteq_{\text{MAX}} n \quad \Longrightarrow \quad n \in \mathsf{workset}^*$$

Obviously, upon termination, I have $<_{\mathsf{workset}} = <$.

Also, the algorithm always terminates: Observe that the choice $a \in \text{lca}_<(S)$ made by Algorithm 3 is such that $<$-cycles, once established, remain stable, i.e.:

- whenever $\mathsf{IMDOM}[x]$ is about to be updated from some $z \neq \bot$ to z', i have: $\neg z' <^* x$

Also, once $\mathsf{IMDOM}[x] = z \neq \bot$, future changes of $\mathsf{IMDOM}[x]$ to z' are only possible if x is just outside some $<$-cycle Z, and z, z' are in that cycle, i.e.: $z <^* z' <^* z$.

But there cannot be an infinite number of such changes, since this would require there to be some nodes $x_1 \neq x_2$[1], $x_i \notin Z$, $\mathrm{ipdom}_{\sqsupseteq_{\mathrm{MAX}}}(x_i) = Z$ such that

- $\exists x_2 \to_G y_2.\ y_2 <^* x_1$
- $\exists x_1 \to_G y_1.\ y_1 <^* x_2$

But from $y_2 <^* x_1$ i obtain a path

$$\underbrace{y_2, \ldots,}_{\pi_2} x_1 \text{ in } G \text{ with } x_1 \notin \pi_2, \text{ and also: } \pi_2 \cap Z = \emptyset,$$

since if I had such a z, I would also have a cycle $\pi_z = z, \ldots, z', \ldots, z$ in G (for any $z' \in Z$) with $x_1 \notin \pi_z$, because $\neg\, z <^* x_1$ (and: $\neg\, z' <^* x_1$) and hence $\neg\, x_1 \sqsupseteq_{\mathrm{MAX}} z$ (and $\neg\, x_1 \sqsupseteq_{\mathrm{MAX}} z'$). But this contradicts $y_2 <^* x_1$. Similarly, I obtain a path

$$\underbrace{y_1, \ldots,}_{\pi_1} x_2 \text{ in } G \text{ with } x_2 \notin \pi_1, \text{ and also: } \pi_2 \cap Z = \emptyset.$$

But then, the concatenation of these paths form a G cycle without nodes from Z, in contradiction with $\mathrm{ipdom}_{\sqsupseteq_{\mathrm{MAX}}}(x_i) = Z$.

□

[1] or: a sequence $x_1, \ldots, x_n$ in a similair situaion

A.3 Order Dependence

Proof of Lemma 6.1.2 on page 67: Assume $n \rightarrow_{\text{dod}} (m_1, m_2)$.

(i) I show $m_2 \sqsupseteq_{\text{MAX}} m_1$. Let $\pi_1 = m_1, \ldots$ be a maximal path. Let n_l be some successor of n in accord with clause (b) of Definition 6.1.2. By definition (clause (a)) I obtain a path $\pi_n = n, n_l, \ldots, m_1$, and n_l was chosen s.t. $\neg m_2 \in \pi_n$. Again by clause (a), any extension π of π_n to a maximal path $\pi_n\pi$ must contain m_2 (in it's extension π), and I am done (i.e.: $m_2 \in \pi$) for $\pi = \pi_1$.

(ii) First, i show $m \sqsupseteq_{\text{MAX}} m_1$. Assume the opposite. Then I have a maximal path $\pi_1 = m_1, \ldots$ with $m \notin \pi_1$. I note that, since $m_2 \sqsupseteq_{\text{MAX}} m_1$ (see (i)), I have

$$\pi_1 = m_1, \ldots, m_2, \pi_1' \quad \text{with } m \notin \pi_1' \tag{A.1}$$

From $m \notin \pi_1$ I infer that

$$\forall \pi_n = n, \ldots, m_1.\ m \in \pi_n \tag{A.2}$$

since otherwise I had a maximal path $\pi_n = n, \ldots, \pi_1$ with $m \notin \pi_n$, in contradiction to $m \sqsupseteq_{\text{MAX}} n$. Specifically, since $n \neq m$, given n_l as in clause (b), for any path

$$\pi_l = n_l, \ldots, m_1$$

with only one appearance of m_1, I have $m \in \pi_l$ and $m_2 \notin \pi_l$. Then, given n_r as in clause (c), for any path

$$\pi_r = n_r, \ldots, m_2$$

with only one appearance of m_2 I have $m_1 \notin \pi_r$ and $m \notin \pi_r$, since if I had $m \in \pi_r$, I'd have a path $n_r, \ldots, m, \ldots, m_1$ that does *not* contain m_2, in contradiction of clause (c).
Now, since $m_1 \sqsupseteq_{\text{MAX}} n$ and $m_1 \neq n$, *any* maximal extension $\pi = \pi_r\pi'$ to π_r must contain m_1. But because of (A.2), any such

extension must also contain m, and it must appear in π', which contradicts (A.1), because $\pi_r\pi'_1$ is a maximal extension to π_r.

$m_1 \sqsupseteq_{\text{MAX}} m$ then is obvious from (i), due to the fact that $m_1 \neq m_2$, and hence the set $\{ m \mid m \sqsupseteq_{\text{MAX}} m_1 \}$ form a cycle (remember that for any transitive reduction $>_{\text{MAX}}$ of $\sqsupseteq_{\text{MAX}}$, the graph $<_{\text{MAX}}$ is a psuedo-forest).

(iii) I show $\neg\, m_1 \rightarrow^*_G n$. Assume the opposite. Given n_l as in clause (b), for any path

$$\pi_l = n_l, \ldots, m_1$$

with only one appearance of m_1 I have $m_2 \notin \pi_l$. From $m_1 \rightarrow^*_G n$, I infer that any maximal path starting in m_1 must contain m_2 before n since otherwise, I would have a cycle

$$n_l, \ldots, m_1, \ldots, n, n_l$$

not containung m_2, contradicting $m_2 \sqsupseteq_{\text{MAX}} n$ (clause (a)).

But with $m_1 \rightarrow^*_G n$ and $m_2 \rightarrow^*_G m_1$ (see (i)) I also have $m_2 \rightarrow^*_G n$, and likewise infer that any maximal path starting in m_2 must contain m_1 before n. But then, any maximal path starting in m_1 must be of the form

$$\underbrace{m_1, \ldots, m_2, \ldots, m_1, \ldots, m_2, \ldots}_{n \notin}$$

contradicting $m_1 \rightarrow^*_G n$.

□

B Nontermination (In-)Sensitive Control Dependence

All your life people will tell you things. And most of the time, probably ninety-five percent of the time, what they'll tell you will be wrong.

(Michael Crichton — The Lost World)

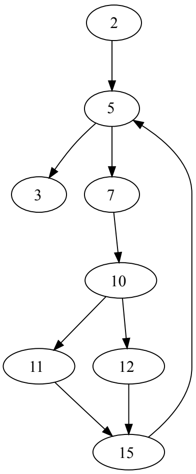
B.1 Analysis of previous Algorithms

Ranganath et al. propose Algorithm 14 for the computation of $\rightarrow$ntscd[1]. Their algorithm works by computing — for each $x \in N, n \in \text{COND}_G$ – the set

$$\mathsf{S}\,[x,n] = \{\, n \rightarrow_G m \mid x \sqsupseteq_{\text{SINK}} m \,\}$$

of edges $n \rightarrow_G m$ starting in n such that every maximal path starting in m contains x. The authors also write t_{nm} for the edge $n \rightarrow_G m$.

I added to their algorithm the highlighted parts, which are missing in [Ran+07]. In order to see that these are indeed necessary, consider the graph depicted on the right. Note that it has the unique exit node $n_x = 3$, and that 15 is standard (and hence also both nontermination sensitively and insensitively) control-dependent on 5, so we also expect $5 \rightarrow_{\text{ntscd}} 15$. Specifically, we need $\mathsf{S}\,[15,5] = \{t_{5,7}\}$.



If the highlighted parts are missing, the sequence $n \in [3,7,10,11,12,15,5]$ is a possible iteration order. I cannot learn that $t_{5,7} \in \mathsf{S}[15,5]$ before I learn that both $t_{10,11}, t_{10,12} \in \mathsf{S}[15,10]$, which I do in the iteration $n = 12$. I do not learn $t_{5,7} \in \mathsf{S}[15,5]$ at that iteration because I did not (and must not!) learn that $t_{5,7} \in \mathsf{S}[12,5]$. But I do not learn $t_{5,7} \in \mathsf{S}[15,5]$ in iteration $n = 15$ (because 15 is not a conditional node, and not $15 \rightarrow_G 15$) or $n = 5$ (trivially), either.
If, on the other hand, in iteration $n = 12$ I put $p = 10$ in the workset, I learn $t_{5,7} \in \mathsf{S}[15,5]$ from $|\mathsf{S}[15,12]| = |\{11,12\}| = 2$ in a later iteration $n = 10$.

[1] For purposes of comparison only, I temporarily adapt adapt their choice of variable naming.

```
Input  : A CFG $G = (N, E)$
Output: A map $\mathsf{NTSCD}$ such that $\mathsf{NTSCD}[n] = \{m \mid n \rightarrow_{\text{ntscd}} m\}$
begin
    for $n \in \text{COND}_G,\ n \rightarrow_G m$ do
        $\mathsf{S}[m, n] \leftarrow \{t_{nm}\}$
        $\mathsf{workset} \leftarrow \mathit{workset} \cup \{m\}$
    end
    while $\mathsf{workset} \neq \emptyset$ do
        $n \leftarrow \mathit{remove}(\mathsf{workset})$
        if $\{m \mid n \rightarrow_G m\} = \{m\}$ then
            for $p \in \text{COND}_G, \mathsf{S}[n, p] \setminus \mathsf{S}[m, p] \neq \emptyset$ do
                $\mathsf{S}[m, p] \leftarrow \mathsf{S}[m, p] \cup \mathsf{S}[n, p]$
                $\mathsf{workset} \leftarrow \mathsf{workset} \cup \{m\}\ \boxed{\cup \{p\}}$
            end
        end
        if $\{m \mid n \rightarrow_G m\} = \{m_1, m_2, \ldots\}$ then
            for $m \in N$ do
                if $|\mathsf{S}[m, n]| = |\{x \mid n \rightarrow_G x\}|$ then
                    for $p \in \text{COND}_G \setminus \{n\},\ \mathsf{S}[n, p] \setminus \mathsf{S}[m, p] \neq \emptyset$ do
                        $\mathsf{S}[m, p] \leftarrow \mathsf{S}[m, p] \cup \mathsf{S}[n, p]$
                        $\mathsf{workset} \leftarrow \mathsf{workset} \cup \{m\}\ \boxed{\cup \{p\}}$
                    end
                end
            end
        end
    end
    for $n \in N,\ m \in \text{COND}_G$ do
        if $0 < |\mathsf{S}[m, n]| < |\{x \mid n \rightarrow_G x\}|$ then
            $\mathsf{NTSCD}[m] \leftarrow \mathsf{NTSCD}[m] \cup \{n\}$
        end
    end
end
```

Algorithm 14: An Algorithm for $\rightarrow_{\text{ntscd}}$. The highlighted parts were missing in [Ran+07], but are crucial for correctness.

Is Algorithm 14 as given here now in fact correct? For now, I can at least make use of a trusted algorithm for $\sqsupseteq_{\text{POST}}$-control dependence, and validate the second implication of Theorem 4.1.1 empirically:

Observation B.1.1. Let NTSCD be computed via Algorithm 14, ans G be a CFG with unique exit node. Then

$$n \rightarrow_{\text{cd}} m \implies m \in \mathsf{NTSCD}[n]$$

Indeed with the previously missing workset updates in place, Algorithm 14 is a standard workset implementation of a least fixed point computation:

Observation B.1.2. Algorithm 14 computes the least fixed point of the monotone functional implicit in the rule system S_4 below:

$$\frac{p \rightarrow_G x}{(p,x) \in \mathsf{S}[x,p]}\,\mathsf{S}_4^{\text{suc}} \qquad \frac{\{m \mid n \rightarrow_G m\} = \{m\} \qquad (p,x) \in \mathsf{S}[n,p]}{(p,x) \in \mathsf{S}[m,p]}\,\mathsf{S}_4^{\text{lin}}$$

$$\frac{n \neq p \qquad |\mathsf{S}[m,n]| = |\{y \mid n \rightarrow_G y\}| \qquad (p,x) \in \mathsf{S}[n,p]}{(p,x) \in \mathsf{S}[m,p]}\,\mathsf{S}_4 cond$$

Here, S is understood to be a map $(N \times \text{COND}_G) \rightarrow 2^E$.

Lemma B.1.1. S_4 is *sound*, i.e.

$$\mathsf{S}_4 \vdash (p,x) \in \mathsf{S}[m,p] \implies m \sqsupseteq_{\text{MAX}} x$$

But S_4 also is *complete* (i.e.: the reversed implication holds for edges $p \rightarrow_G x$) because any fact $m \sqsupseteq_{\text{MAX}} x$ necessarily has a *finite* proof (in S_4).

Proof (Sketch): Soundness follows directly by induction. For completness, I assume $m \sqsupseteq_{\text{MAX}} p$. For conditional nodes $p \neq m$ this means: $m \sqsupseteq_{\text{MAX}} x$ for all x s.t. $p \rightarrow_G x$. Then either m appears on all the linear segments starting in x — in which case rule $\mathsf{S}_4^{\text{suc}}$ and $\mathsf{S}_4^{\text{lin}}$ suffice, and I stop with a finite proof —, or there exists successors x of p such that from x I can reach the next conditional node p' without visiting m. For such x, I continue to search for proofs that $m \sqsupseteq_{\text{MAX}} p'$. This search must eventually come to an end, because otherwise, I would have found a sequence $p, p', \ldots, p'$ of conditional nodes such that m does not necessarily appear on every paths π through $p, p', \ldots, p'$, and I could construct a maximal path $\pi, \pi, \ldots$ without m, in contradiction of the assumption $m \sqsupseteq_{\text{MAX}} p$.

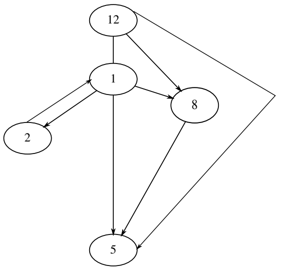
Ranganath et al. also claim an algorithm for the computation of $\rightarrow_{\text{nticd}}$. I repeat it — together with the fixes w.r.t workset management I proposed for Algorithm 14 — as Algorithm 15. In contrast to Algorithm 14, that algorithm is *not* correct, and there appears to be not obvious fix. First observe that — if we allow the additional highlighted lines, and then again read it as a work-set algorithm that computes the least fixed point of some monotone functional — Algorithm 15 differs from Algorithm 14 only by adding the rule

$$\frac{p \rightarrow_G m \qquad (p,x) \in \mathsf{S}\,[p,p]}{(p,x) \in \mathsf{S}\,[m,p]}\,\mathsf{S}_5^{\text{self}}$$

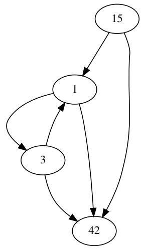
to the rule system S_4. Let the resulting system be denoted by S_5.

```
Input  : A CFG $G = (N, E)$
Output: ~~A map NTICD such that $\mathsf{NTICD}[n] = \{m \mid n \to_{\text{ntscd}} m\}$~~
begin
    for $n \in \text{COND}_G,\ n \to_G m$ do
        $\mathsf{S}[m,n] \leftarrow \{t_{nm}\}$
        $\mathsf{workset} \leftarrow \mathit{workset} \cup \{m\}$
    end
    while $\mathsf{workset} \neq \emptyset$ do
        $n \leftarrow \mathit{remove}(\mathsf{workset})$
        if $\{m \mid n \to_G m\} = \{m\}$ then
            . . . same as Algorithm 14 . . .
        end
        if $\{m \mid n \to_G m\} = \{m_1, m_2, \ldots\}$ then
            . . . same as Algorithm 14 . . .
        end
        if $|\mathsf{S}[n,n]| > 0$ then
            for $n \to_G m,\ m \neq n,\ \mathsf{S}[n,n] \setminus \mathsf{S}[m,n] \neq \emptyset$ do
                $\mathsf{S}[m,n] \leftarrow \mathsf{S}[m,n] \cup \mathsf{S}[n,n]$
                $\mathsf{workset} \leftarrow \mathsf{workset} \cup \{m\}$ $\boxed{\cup \{p\}}$
            end
        end
    end
    for $n \in N,\ m \in \text{COND}_G$ do
        if $0 < |\mathsf{S}[m,n]| < |\{x \mid n \to_G x\}|$ then
            $\mathsf{NTICD}[m] \leftarrow \mathsf{NTICD}[m] \cup \{n\}$
        end
    end
end
```

Algorithm 15: An incorrect Algorithm for $\to_{\text{nticd}}$. Only the $\boxed{\text{framed}}$ part is new (w.r.t. Algorithm 14). The $\boxed{\text{highlighted}}$ parts were missing in [Ran+07]. The algorithm is incorrect no matter if these are present or not.



(a) Unsoundness.



(b) $\rightarrow_{\text{nticd}}$ requires circular reasoning.

Figure B.1: Problems with Algorithm 15.

Unfortunately, this rule is not even *sound*, i.e. it does *not* hold that

$$\mathsf{S}_5 \vdash (p, x) \in \mathsf{S}[m, p] \quad \Longrightarrow \quad m \sqsupseteq_{\text{SINK}} x$$

In order to see this, consider the CFG in Figure B.1a. Note that I expect from a solution S:

$$\mathsf{S}[1,1] = \{(1,2)\} \quad \text{but } \textit{not}\text{: } (1,2) \in \mathsf{S}[8,1]$$

because any (sink-bound) path from 2 does contain 1, but there is the sink-bound path $2, 1, 5$ from 2 which does *not* contain 8. But rule $\mathsf{S}_5^{\text{self}}$ allows me to infer just that!

This problem with rule system S_5 does not yet necessarily lead to wrong results *in the resulting map* NTICD. However, a much more serious problem is examplified in Figure B.1b. Here, Algorithm 15 computes

$$\mathsf{S}[42,1] = \{(1,42)\} \qquad \text{and concludes:} \qquad 1 \rightarrow_{\text{nticd}} 42$$

while in reality:

$$\mathsf{S}[41,1] = \{(1,42),(1,3)\} \quad \text{and} \qquad \neg\, 1 \rightarrow_{\text{nticd}} 42$$

To see that in fact $(1,3) \in \mathsf{S}\,[42,1]$ (i.e.: $42 \sqsupseteq_{\textsc{sink}} 3$), just note that $\{42\}$ is the *only* control sink.

Can Algorithm 15 be fixed by adding another rule to S_5? To answer this, I explain how I fail to infer $(1,3) \in \mathsf{S}\,[42,1]$ using Algorithm 15 (i.e.: how I fail to infer $\mathsf{S}_5 \vdash (1,3) \in \mathsf{S}\,[42,1]$ in the rule system S_5). If I am to infer $\mathsf{S}_5 \vdash (1,3) \in \mathsf{S}\,[42,1]$, I need either to infer $\mathsf{S}_5 \vdash (1,3) \in \mathsf{S}\,[1,1]$ via rule $\mathsf{S}_5^{\mathsf{self}}$, or

$$|\mathsf{S}\,[42,n]| = |\{y \mid n \to_G y\}| \text{ and } (1,3) \in \mathsf{S}\,[n,1]$$

for some $n \in \textsc{cond}_G, n \neq 1$. Since (in order to apply rule $\mathsf{S}_4\mathit{cond}$) attempting to infer $\mathsf{S}_5 \vdash (1,3) \in \mathsf{S}\,[15,1]$ is pointless , I am left with $n = 3$, and I am required to show $\mathsf{S}_5 \vdash (3,42) \in \mathsf{S}\,[42,3]$ (trivial) *and* $\mathsf{S}_5 \vdash (3,1) \in \mathsf{S}\,[42,3]$. Then, I need either to infer $\mathsf{S}_5 \vdash (3,1) \in \mathsf{S}\,[3,3]$ via rule $\mathsf{S}_5^{\mathsf{self}}$, or

$$\left|\mathsf{S}\,[42,n']\right| = \left|\{y \mid n' \to_G y\}\right| \text{ and } (3,1) \in \mathsf{S}\,[n',3]$$

for some $n' \in \textsc{cond}_G, n' \neq 3$. Since attempting to infer $\mathsf{S}_5 \vdash (3,1) \in \mathsf{S}\,[15,3]$ is pointless, I am left with $n' = 1$, and I am required to show $\mathsf{S}_5 \vdash (1,42) \in \mathsf{S}\,[42,1]$ (trivial) *and* $\mathsf{S}_5 \vdash (1,3) \in \mathsf{S}\,[42,1]$.

In summary, I need to complete one of the following proof trees:

$$\text{(a)}\left\{\ \dfrac{\dfrac{?}{(1,3) \in \mathsf{S}\,[1,1]}}{(1,3) \in \mathsf{S}\,[42,1]} \qquad \dfrac{\cdots \quad \dfrac{\dfrac{?}{(3,1) \in \mathsf{S}\,[3,3]}}{(3,1) \in \mathsf{S}\,[42,3]} \quad \cdots}{(1,3) \in \mathsf{S}\,[42,1]}\ \right\}\text{(b)}$$

$$\left.\dfrac{\cdots \qquad \dfrac{\dfrac{?}{\cdots \quad (1,3) \in \mathsf{S}\,[42,1] \quad \cdots}}{(3,1) \in \mathsf{S}\,[42,3]} \qquad \cdots}{(1,3) \in \mathsf{S}\,[42,1]}\right\}\text{(c)}$$

The last proof tree (c) is circular, so at most the first two are feasible. But attempting $\mathsf{S}_5 \vdash (1,3) \in \mathsf{S}\,[1,1]$, the only sensible rule is S_4*cond* with $n = 3$, which requires me to infer $\mathsf{S}_5 \vdash (3,1) \in \mathsf{S}\,[1,3]$ (trivial) *and* $\mathsf{S}_5 \vdash (3,42) \in \mathsf{S}\,[1,3]$, which I cannot because it is false. Similarly, attempting $\mathsf{S}_5 \vdash (3,1) \in \mathsf{S}\,[3,3]$, I have to use S_4*cond* with $n = 1$, but this requires me to infer $\mathsf{S}_5 \vdash (1,3) \in \mathsf{S}\,[3,1]$ (trivial) *and* $\mathsf{S}_5 \vdash (1,42) \in \mathsf{S}\,[3,1]$, which I cannot.

I have demonstrated how the rule system S_5 is incapable of finitely proving $(1,3) \in \mathsf{S}\,[42,1]$, and thus how Algorithm 15 is incorrect. How can I find a correct algorithm? Refocus on the last "circular" proof tree (c), and recall how I argued for the completeness of S_4 w.r.t $\sqsupseteq_{\text{MAX}}$: I demonstrated that whenever $m \sqsupseteq_{\text{MAX}} x$, I could find a *finite proof* of this in S_4. Now, $\sqsupseteq_{\text{SINK}}$ differs from $\sqsupseteq_{\text{MAX}}$ in that it disregards infinite loops such as the loop such as $1 \rightarrow_G 3 \rightarrow_G 1$. In particular, considering *sink-bound* paths starting in x instead of maximal paths starting in x, whenever $\neg m \sqsupseteq_{\text{SINK}} x$, I must be able to find a *finite disproof*, disregarding loops on paths towards x. This suggests that we should believe $(1,3) \in \mathsf{S}\,[42,1]$ since, as considering the circular proof tree (c), it cannot be *finitely disproven*, i.e.: we should define S *co*-inductively, by taking the *greatest* fixed point of some monotone functional (implicit in some rule-system).

Is the rule system S_4 (or: S_5) a suitable rule system for the computation of $\rightarrow_{\text{nticd}}$ when read as a *co*-inductive definition of S? Not quite, since it suffers from three problems.

1. It does *not* enforce that from $(p,x) \in \mathsf{S}\,[m,p]$, it follows that: $x \rightarrow_G^* m$.

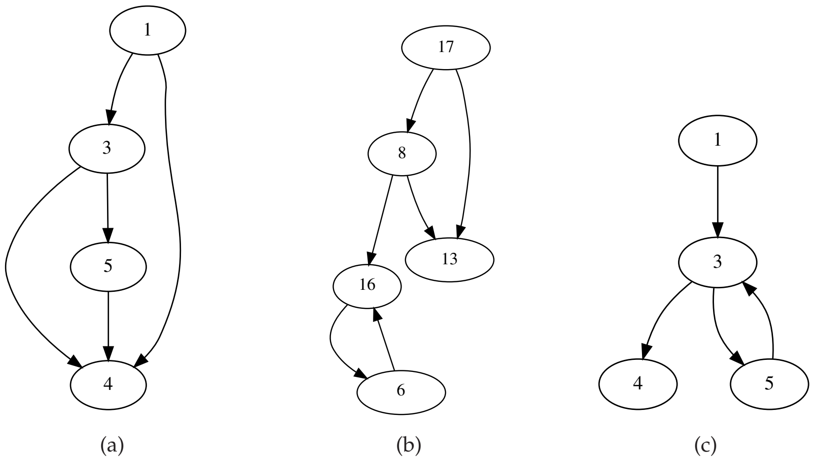


Figure B.2:

2. Following the graph structure (from n to m along $n \to_G m$) in rules $\mathsf{S}_4 cond$ and $\mathsf{S}_4^{\mathrm{lin}}$ one step a time is too liberal, allowing too many “self-justifications”.

3. The requirement $n \neq p$ in $\mathsf{S}_4 cond$ is too strict (while, as i have shown, $\mathsf{S}_5^{\mathrm{self}}$, is too liberal).

The problem 1 is demonstrated in the CFG from Figure B.2a. I cannot finitely disprove in S_4 the following assertions, of which only the highlighted is true w.r.t $\to_{\mathrm{nticd}}$, and all the other assertions $(p,x) \in \mathsf{S}[m,p]$ are false because $\neg\, x \to_G^* m$.

$$\begin{array}{ll}
(3,4),(3,5) & \in \mathsf{S}[1,3] \\
(1,3),(1,4) & \in \mathsf{S}[1,1] \\
\boxed{(1,3)},(1,4) & \in \mathsf{S}[3,1] \\
(3,4),(3,5) & \in \mathsf{S}[3,3]
\end{array}$$

as demonstrated by the following partial derivation trees consistent with S

$$\frac{(1,3),(1,4) \in \mathsf{S}\,[1,1] \qquad (3,4) \in \mathsf{S}\,[1,3]}{(3,4) \in \mathsf{S}\,[1,3]}$$

$$\frac{(3,4),(3,5) \in \mathsf{S}\,[1,1] \qquad (3,5) \in \mathsf{S}\,[1,3]}{(3,5) \in \mathsf{S}\,[1,3]}$$

$$\frac{(3,4),(3,5) \in \mathsf{S}\,[1,3] \qquad \dfrac{}{(1,3) \in \mathsf{S}\,[3,1]}\,\mathsf{S}_4^{\mathrm{suc}}}{(1,3) \in \mathsf{S}\,[1,1]}$$

$$\frac{(3,4),(3,5) \in \mathsf{S}\,[1,3] \qquad (1,4) \in \mathsf{S}\,[3,1]}{(1,4) \in \mathsf{S}\,[1,1]}$$

$$\frac{(3,4),(3,5) \in \mathsf{S}\,[3,3] \qquad (1,4) \in \mathsf{S}\,[3,1]}{(1,4) \in \mathsf{S}\,[3,1]}$$

$$\frac{\mathsf{S}_4^{\mathrm{suc}}\,\dfrac{}{(1,3) \in \mathsf{S}\,[3,1]} \qquad (1,4) \in \mathsf{S}\,[3,1] \qquad (3,4) \in \mathsf{S}\,[1,3]}{(3,4) \in \mathsf{S}\,[3,3]}$$

From such a S, I would have to conclude: $\neg 1 \rightarrow_{\mathrm{nticd}} 3$, when in reality $(1,3) \in \mathsf{S}\,[3,1]$ but *not* $(1,4) \in \mathsf{S}\,[3,1]$, and hence $1 \rightarrow_{\mathrm{nticd}} 3$.

The problem 2 is demonstrated in Figure B.2b. I cannot finitely disprove in S_4 the following false assertions

$$\begin{aligned} (17,8) &\in \mathsf{S}\,[16,17] \\ (17,8) &\in \mathsf{S}\,[6,17] \end{aligned}$$

as demonstrated by the following partial derivation trees consistent with S

$$\mathsf{S}_4^{\mathrm{lin}}\,\frac{(17,8) \in \mathsf{S}\,[6,17]}{(17,8) \in \mathsf{S}\,[16,17]}$$

$$\mathsf{S}_4^{\text{lin}}\ \frac{(17,8) \in \mathsf{S}\,[16,17]}{(17,8) \in \mathsf{S}\,[6,17]}$$

which is valid even if I — in the light of the *previous* example — , also require

$$(p,x) \in \mathsf{S}\,[m,p] \quad \Longrightarrow \quad x \to_G^* m \tag{B.1}$$

But then I *can* disprove, e.g., $(17,13) \in \mathsf{S}\,[6,17]$ in "$\mathsf{S}_4 + (\text{B.1})$", and falsely conclude $17 \to_{\text{nticd}} 6$ (what holds here is: $17 \to_{\text{nticd}} 8 \to_{\text{nticd}} 6$).

Note that the problem with rule $\mathsf{S}_4^{\text{lin}}$ here is that it allows me to "self-validate" the cycle $(17,8) \in \mathsf{S}\,[6,17]\,,\mathsf{S}\,[8,17]$ for paths starting in the edge $17 \to_G 8$ *without* requiring me to validate that from node 8 (i.e: the "next" conditional after 17 for paths starting in $17 \to_G 8$), any sink-path reaches that cycle. Similarly, $\mathsf{S}_4^{\text{lin}}$ allows me to validate an assertion $(p,x) \in \mathsf{S}\,[m,p]$ using *any* other conditional node n (reachable from x), without requiring me to validate

$$\left|\mathsf{S}\,[m,n']\right| = \left|\{y \mid n' \to_G y\}\right|$$

for those conditional nodes n' "in between" x an m.

The problem 3 is demonstrate in Figure B.2c. Here, I can disprove the following *true* fact in S_4:

$$(3,5) \quad \in \mathsf{S}\,[4,3]$$

and erroneously conclude $3 \to_{\text{nticd}} 4$.

B.2 Duality of Nontermination (In-)Sensitivity

Discussing the example in Figure B.1b, I concluded that for the computation of $\rightarrow_{\text{nticd}}$, I need to characterize S co-inductively, rather than inductively. I then proceeded by explaining why the rule set S_4 is not suitable for such a characterization. Now in this section, I will propose a rule system S_3 (and also write S_3 for the corresponding monotone functional) such that

- for $\mathsf{S} = \nu \mathsf{S}_3$
$$\mathsf{S}\,[m,p] = \{\, p \rightarrow_G x \mid m \sqsupseteq_{\text{SINK}} x \,\}$$
and *at the same time*:
- for $\mathsf{S} = \mu \mathsf{S}_3$
$$\mathsf{S}\,[m,p] = \{\, p \rightarrow_G x \mid m \sqsupseteq_{\text{MAX}} x \,\}$$

Informally:

nontermination insensitivity	=	greatest fixed point
nontermination sensitivity	=	least fixed point

Before I introduce this system S_3 of rules, I go on a short detour in which I explain this correspondence from the point of view of *safety* and *liveness* properties. It is well-known that liveness properties correspond to least fixed-points, while safety-properties correspond to greatest fixed points. In fact, adopting the standard definitions of *liveness* and *safety* to executions I obtain:

Definition B.2.1 ([AS85]). A set $\mathcal{S} \subseteq N^\omega$ (of sequences called *inifinite traces*) is called a *safety* property iff a violation of it is finitely observable and irremediable, i.e. iff whenever $\pi \notin \mathcal{S}$ for some inifinite trace π, there already exists some *finite* prefix π_0 of π such that:

- for all infinite traces $\pi' = \pi_0 \ldots \in N^\omega$ extending π_0: $\pi' \notin \mathcal{S}$

Definition B.2.2 ([AS85])**.** A set $\mathcal{L} \subseteq N^\omega$ (of inifinite traces) is called a *liveness* property iff it is always possible (and possibly infinite), i.e. iff for *any* finite trace π_0, there already exists some *infinite* extension $\pi' = \pi_0 \ldots$ of π_0 such that

- $\pi' \in \mathcal{L}$

Note that in this terminology, *traces* are in fact only arbitrary sequences of nodes, not necessarily corresponding to proper paths in some graph G.

Definition B.2.3. I write ${}_xN \subseteq N^\omega$ for the set of infinite traces starting in x. Moreover, let $G = (N, E)$ be some CFG, and $x \to^*_G m$. Then i define

$$\Pi^G_{\text{SINK}}[x,m] = $$
$$\left\{\pi \in {}_xN \;\middle|\; \neg\left(\pi \text{ has a prefix } \pi_0 = x \ldots n \text{ with } m \notin \pi_0, \neg\, n \to^*_G m\right)\right\}$$

$$\Pi^G_{\text{MAX}}[x,m] = $$
$$\{\pi \in {}_xN \mid \neg(\qquad\qquad\qquad\qquad m \notin \pi \qquad\qquad)\}$$

Intuitively, any trace $\pi \notin \Pi^G_{\text{SINK}}[x,m]$ is a counterexample to the claim that node x nontermination insensitively postdominates node m, while any maximal trace $\pi \notin \Pi^G_{\text{MAX}}[x,m]$ is a counterexample to the claim that node x nontermination sensitively postdominates node m. Also intuitively, $\Pi^G_{\text{SINK}}[x,m]$ is a safety property, since I can just expose the prefix π_0 obtained by definition. $\Pi^G_{\text{MAX}}[x,m]$ is trivially a liveness property, since any finite sequence of nodes can be extended to include m (what's not clear is whether this can be done with a proper *path* in some graph G, i.e.: whether this liveness property holds in G).

In order to establish this formally, I make a technical modification to the underlying graph G.

Observation B.2.1. Let $G_0 = (N, E)$ be some CFG, $x, m \in N$ some nodes, $x \rightarrow_G^* m$, and define

$$G = (N, E \cup \{\, (n_x, n_x) \mid n_x \text{ is an exit node} \,\})$$

Then the maximal paths in G are exactly the infinite paths, and

1. $\Pi^G_{\text{SINK}}[x, m]$ is a safety property.
2. $\Pi^G_{\text{MAX}}[x, m]$ is a liveness property.
3. $\forall \pi \in {}_x\Pi_{\text{SINK}}.\ m \in \pi \quad \Longleftrightarrow \quad {}_x\Pi^G \cap N^\omega \subseteq \Pi^G_{\text{SINK}}[x, m]$
4. $\forall \pi \in {}_x\Pi_{\text{MAX}}.\ m \in \pi \quad \Longleftrightarrow \quad {}_x\Pi^G \cap N^\omega \subseteq \Pi^G_{\text{MAX}}[x, m]$

Proof (Sketch):

1. Let $\pi \notin \Pi^G_{\text{SINK}}[x, m]$. The case $\pi \notin {}_xN$ is trivial, so i can assume some prefix $\pi_0 = x \ldots n$ of π s.t. $m \notin \pi_0$ and $\neg\, n \rightarrow_G^* m$. But this then also holds for every π' extending π_0, so $\pi' \notin \Pi^G_{\text{SINK}}[x, m]$.
2. This is trivially true.
3. First, I show the implication $\Rightarrow$.

   Let $\pi = x \ldots$ be some infinite path starting in x. If it is a sink path, then $m \in \pi$, hence $\pi \in \Pi^G_{\text{SINK}}[x, m]$. Otherwise, let let $\pi_0 = x \ldots n$ of π be any prefix of π. I can always extend this in G to *obtain* a sink-bound path π' starting with π_0. But then $m \in \pi'$, so either $m \in \pi_0$, or $n \rightarrow_G^* m$.

   Turning to the implication $\Leftarrow$, assume ${}_x\Pi^G \cap N^\omega \subseteq \Pi^G_{\text{SINK}}[x, m]$, and $\pi \in {}_x\Pi_{\text{SINK}}$. Let $\pi_0 = x \ldots n$ be some prefix of π s.t. $n \in S$, with S being the sink that π is bound for. Then either $m \in \pi_0$, and I am done, or $n \rightarrow_G^* m$. But then $m \in S$, and $m \in \pi$, because a sink-bound path visits every $m \in S$ infinitely often.
4. Trivial, given that the maximal paths in G are exactly the infinite paths.

But, again informally,

$$\begin{aligned}\text{liveness} &= \text{least fixed point}\\ \text{safety} &= \text{greatest fixed point}\end{aligned}$$

which is made somewhat more explicit in the setting of the modal μ calculus (see, e.g., [BS07]). So the fact that $\rightarrow_{\text{nticd}}$ is obtainable via greatest fixed points, and $\rightarrow_{\text{ntscd}}$ via least fixed points is — *a posteriori* — perhaps not so surprising, while perhaps the fact that it is obtained from a single functional is.

Remark B.2.1. My characterization of $\sqsupseteq_{\text{MAX}}$ as liveness-, and $\sqsupseteq_{\text{SINK}}$ as safety property is unusual, since the definitions involve the predicate $n \rightarrow^{*}_{G} m$, which is a property of the graphs G structure, while liveness-, safety properties usually only involve predicates on the graphs nodes (i.e.: states).

Note also that i did *not* characterize $\rightarrow_{\text{nticd}}$ or $\rightarrow_{\text{nticd}}$ as trace properties. They presumably are not trace properties, but *2-hyper-properties*[CS10].

Returning form this diversion, I will now finally define the rules S3. First, I need a concept already implicit in the argument for Lemma B.1.1.

Definition B.2.4. Given any node x, it's *next conditional node*

$$\text{next}_{\text{COND}}\left[x\right] = \epsilon\ n.\ n \in \text{COND}_{G}, x \rightarrow^{\pi}_{G} n, \pi \cap \text{COND}_{G} = \{n\}$$

is *the* (if any!) unique conditional node n that is reachable from x without first reaching another conditional node.

The set of nodes on the path to the next conditional node from x (or just the nodes linearly following x if there is no such node) is:

$$\overline{\text{next}_{\text{COND}}}\left[x\right] = \begin{cases} \{\, m \mid x \rightarrow^{\pi}_{G} n,\ \pi \text{ cycle-free},\ m \in \pi \,\} & \text{next}_{\text{COND}}\left[x\right] = n \\ \{\, m \mid x \rightarrow^{*}_{G} m \,\} & \text{next}_{\text{COND}}\left[x\right] = \bot \end{cases}$$

Similarly,

$$\text{prev}_{\text{COND}}[x] = \{\, p \mid p \in \text{COND}_G, p \rightarrow^{\pi}_{G} x, \pi \cap \text{COND}_G = \{p\}, \pi \neq x \,\}$$

is the set of set of conditional nodes p that can reach x (in at least one step) without first reaching another conditional node.

Observation B.2.2. Note that for conditional nodes $x \in \text{COND}_G$

- $\text{next}_{\text{COND}}[x] = x$, but
- *not necessarily* $x \in \text{prev}_{\text{COND}}[x]$.

Remark B.2.2. Both $\text{next}_{\text{COND}}$, $\overline{\text{next}_{\text{COND}}}$ and $\text{prev}_{\text{COND}}$ can easily and (simultaneously) be computed in $\mathcal{O}(|\text{COND}_G| \times |N|)$ steps, but in practice usually not much more than $\mathcal{O}(|\text{COND}_G|)$.

Now, I can state S_3:

Definition B.2.5. With S_3 I denote both the rule system below, and the corresponding monotone functional.

$$\frac{p \rightarrow_G x \qquad m \in \overline{\text{next}_{\text{COND}}}[x]}{(p,x) \in \mathsf{S}[m,p]}\,\mathsf{S}_3^{\text{lin}}$$

$$\frac{p \rightarrow_G x \qquad n = \text{next}_{\text{COND}}[x] \qquad \begin{matrix} x \rightarrow^{*}_{G} m \\ |\mathsf{S}[m,n]| = |\{y \mid n \rightarrow y\}| \end{matrix}}{(p,x) \in \mathsf{S}[m,p]}\,\mathsf{S}_3^{\text{cond}}$$

Here, again, S is understood to be a map $(N \times \text{COND}_G) \rightarrow 2^E$.

Observation B.2.3. With regard to the *least* fixed point, S_3 is equivalent to S_4:

$$\mu\mathsf{S}_3 = \mu\mathsf{S}_4$$

Remark B.2.3. This is also obvious: $\mathsf{S}_3^{\text{lin}}$ subsumes $\mathsf{S}_4^{\text{suc}}$ and $\mathsf{S}_4^{\text{lin}}$, but limits its applicability to nodes n in *regions* directly behind x. But this is no limitation w.r.t $\mu\mathsf{S}_4$, since there, any finite derivation along linear segments of G starting with an edge $p \rightarrow_G x$ at $\mathsf{S}_4^{\text{suc}}$ and continuing with $\mathsf{S}_4^{\text{lin}}$ have to end at the next conditional node $\text{next}_{\text{COND}}[x]$, anyway.

With regard to $\mathsf{S}_3^{\text{lin}}$, first note that the requirement $m \rightarrow_G^* x$ is vacuously true in the least fixed point of S_4. Also, w.r.t least fixed points of S_4, dropping the requirement $n \neq p$ does not change anything, since for $n = p$, this rules then *requires* $(p, x) \in \mathsf{S}[m, p]$ in order to derive $(p, x) \in \mathsf{S}[m, p]$.
Moreover, for $n = \text{next}_{\text{COND}}[x]$, I immediately have $(p, x) \in \mathsf{S}[n, p]$ via $\mathsf{S}_3^{\text{lin}}$, so I can drop that requirement in $\mathsf{S}_3^{\text{cond}}$.
Finally, $\mathsf{S} = \mu\mathsf{S}_4$ has the property that whenever I can proof $(p, x) \in \mathsf{S}[m, p]$ with $\mathsf{S}_4 cond$, then I can already proof

$$|\mathsf{S}[m, n]| = |\{y \mid n \rightarrow y\}|$$

for all conditional nodes n reachable from x (without using $(p, x) \in \mathsf{S}[m, p]$), so limiting myself to $n = \text{next}_{\text{COND}}[x]$ does not inhibit my power. Also, for $\mathsf{S} = \mu\mathsf{S}_4$ I never invoke $\mathsf{S}_4 cond$ with n unreachable from x.

B.3 New Algorithms

In this section, I will provide the — to the best of my knowledge – first correct algorithm for the computation of $\rightarrow_{\text{nticd}}$. Before I proof that it really is enough to compute the *greatest* fixed point of S_3 (Theorem B.3.1), I will first provide the Algorithm 16.

Observation B.3.1. Let S be computed by Algorithm 16. Then

$$\mathsf{S} = \nu \mathsf{S}_3$$

Remark B.3.1. The initialization is correct because for *any* S consistent with S_3, $(p, x) \in \mathsf{S}[m, p']$ implies $p = p'$, $x \rightarrow^*_G m$ and $p \rightarrow_G x$ — hence, I need not start the iteration as high as $\top = (p, x) \mapsto E$. The work-set is managed correctly, since any refutation of $(n, y) \in \mathsf{S}[m, n]$ may invalidate at most those assertions $(p, x) \in \mathsf{S}[m, p]$ for which $n = \text{next}_{\text{COND}}[x]$ (see $\mathsf{S}_3^{\text{cond}}$), but then it is enough to re-inspect those (m, p) s.t. $p \in \text{prev}_{\text{COND}}[n]$[2]. In fact, I only need to re-inspect (m, p) the *first* time i refute *any* $(n, y) \in \mathsf{S}[m, n]$. So in Algorithm 16, I can strengthen the highlighted check $\mathsf{S}'_{mp} \subset \mathsf{S}[m, p]$ by replacing it with

$$|\mathsf{S}[m, p]| = |\{x \mid p \rightarrow_G x\}| \quad \wedge \quad \mathsf{S}'_{mp} \subset \mathsf{S}[m, p]$$

in order to reduce the number of iterations.

Incidentally, since the relevant lattice $\left(\sqsubseteq, N \times \text{COND}_G \hookrightarrow 2^E\right)$ is constructed from subset-lattice $\left(\subseteq, 2^E\right)$, the greatest fixed point computation $\mathsf{S} = \nu \mathsf{S}_3$ can be replaced by a least fixed computation on the *dual* (monotone) functional

$$\overline{\mathsf{S}_3}(\mathsf{S})[m, p] = \overline{\mathsf{S}_3\left((m', p') \mapsto \overline{\mathsf{S}[m', p']}\right)[m, p]} \tag{B.2}$$

[2] note the reversal of varibale names n, p w.r.t the algorithms source code

```
Input     : A CFG $G = (N, E)$
Input     : Maps $\text{next}_{\text{COND}}$ and $\overline{\text{next}_{\text{COND}}}$
Output    : A map $\mathsf{NTICD}$ such that $\mathsf{NTICD}[n] = \{m \mid n \rightarrow_{\text{nticd}} m\}$
Data: A map $\mathsf{S}$ from pairs of nodes to sets of nodes
Data: A workset $\mathsf{workset} \subseteq N \times \text{COND}_G$
Notation: $|n|$ for $|\{\, x \mid n \rightarrow_G x \,\}|$
begin
    for $p \in \text{COND}_G,\ p \rightarrow_G x,\ x \rightarrow_G^* m$ do
        $\mathsf{S}[m, p] \leftarrow \mathsf{S}[m, p] \cup \{(p, x)\}$
    end
    for $p \in \text{COND}_G,\ p \rightarrow_G^* m$ do
        $\mathsf{workset} \leftarrow \mathsf{workset} \cup \{(m, p)\}$
    end
    while $\mathsf{workset} \neq \emptyset$ do
        $(m, p) \leftarrow \mathit{remove}(\mathsf{workset})$
        $\mathsf{S'}_{mp} \leftarrow \{\, (p, x) \mid p \rightarrow_G x,\ m \in \overline{\text{next}_{\text{COND}}}\,[x] \,\}$
              $\cup\ \{\, (p, x) \mid p \rightarrow_G x,\ n = \text{next}_{\text{COND}}\,[x],\ |\mathsf{S}[m, n]| = |n| \,\}$
        assert $\mathsf{S'}_{mp} \subseteq \mathsf{S}[m, p]$
        if $\boxed{\mathsf{S'}_{mp} \subset \mathsf{S}[m, p]}$ then
            $\mathsf{workset} \leftarrow \mathsf{workset} \cup \{\, (m, n) \mid n \in \text{prev}_{\text{COND}}\,[p] \,\}$
        end
    end
    for $n \in N,\ m \in \text{COND}_G$ do
        if $0 < |\mathsf{S}[m, n]| < |n|$ then
            $\rightarrow_{\text{nticd}}[m] \leftarrow \rightarrow_{\text{nticd}}[m] \cup \{n\}$
        end
    end
end
```

Algorithm 16: A correct algorithm for $\rightarrow_{\text{nticd}}$. The highlighted check can be optimized.

where for sets $X \subseteq \{\, (p,x) \mid p \to_G x \,\}$ I denote with $\overline{X}$ its complement w.r.t. $\{\, (p,x) \mid p \to_G x \,\}$. If I extend this notion of complement to maps $\mathsf{S} : N \times \textsc{cond}_G \hookrightarrow 2^E$ via

$$\overline{\mathsf{S}}\,[m,p] = \overline{\mathsf{S}\,[m,p]}$$

then definition (B.2) just reads:

$$\overline{\mathsf{S}_3}\,(\mathsf{S}) = \overline{\mathsf{S}_3\,(\overline{\mathsf{S}})}$$

The following lemma is easily shown along the lines of, e.g. [Fri02], Lemma 20.9.

Lemma B.3.1. The greatest fixed point of S_3 is dual to the least fixed point of $\overline{\mathsf{S}_3}$, i.e.:

$$\nu\mathsf{S}_3 = \overline{\mu\overline{\mathsf{S}_3}}$$

The rule system corresponding to $\overline{\mathsf{S}_3}$ is:

$$\frac{p \to_G x \qquad \neg x \to_G^* m}{(p,x) \in \mathsf{S}\,[m,p]}\ \overline{\mathsf{S}_3}^{\text{unreach}}$$

$$\frac{p \to_G x \qquad \neg m \in \overline{\text{next}_{\text{COND}}}\,[x] \qquad n = \text{next}_{\text{COND}}\,[x] \qquad |\mathsf{S}\,[m,n]| \neq 0}{(p,x) \in \mathsf{S}\,[m,p]}\ \overline{\mathsf{S}_3}^{\text{cond}}$$

To understand the following Lemma, recall the safety-property

$$\{\pi \in {}_xN \mid \neg\,(\pi \text{ has a prefix } \pi_0 = x \ldots n \text{ such that } m \notin \pi_0, \neg\, n \to_G^* m)\}$$

from Definition B.2.3 that characterized $m \sqsupseteq_{\text{SINK}} x$.

Lemma B.3.2. For $\mathsf{S} = \mu\overline{\mathsf{S}_3}$,

$$(p,x) \in \mathsf{S}\,[m,p] \quad\iff\quad \exists n, \pi_0 . m \notin \pi_0 \;\wedge\; x \to_G^{\pi_0} n \;\wedge\; \neg n \to_G^{*} m$$

Proof (Sketch): For the implication $\implies$, proceed by induction on the derivation of $(p,x) \in \mathsf{S}\,[m,p]$. Case $\overline{\mathsf{S}_3}^{\text{unreach}}$ is trivial. For case $\overline{\mathsf{S}_3}^{\text{cond}}$, from $|\mathsf{S}\,[m,n]| \neq 0$ obtain some $n \to_P y$ s.t. — from the induction hypothesis — there exists n', π_0' with $y \to_G^{\pi_0'} n'$, $m \notin \pi_0'$, but *not*: $n' \to_G^{*} m$. But then immediately for

$$\pi_0 = x, \underbrace{\ldots}_{\text{linear}}, n, y, \pi_0'$$

I have: $m \notin \pi_0$, and not: $n' \to_G^{*} m$.

For the implication $\Leftarrow$, assume $m \notin \pi_0 \;\wedge\; x \to_G^{\pi_0} n \;\wedge\; \neg n \to_G^{*} m$. In all but the trivial cases, let $p' \in \text{prev}_{\text{COND}}\,[n]$ be the conditional node immediately preceding n in π_0, and x' the node succeeding that occurrence of p' in π_0. Then — since there are no conditional nodes in G between p' and n — also $\neg p' \to_G^{*} m$, and hence $(p',x') \in \mathsf{S}\,[m,p']$ by rule $\overline{\mathsf{S}_3}^{\text{unreach}}$. But then $(p,x) \in \mathsf{S}\,[m,p]$ by following the path π_0 backwards, applying $\overline{\mathsf{S}_3}^{\text{cond}}$ at each segment $p'' \to_G x''$, $n'' = \text{next}_{\text{COND}}\,[x'']$.

I immediately obtain:

Theorem B.3.1. The rule system S_3 is correct w.r.t it's greatest fixed point, and $\to_{\text{nticd}}$. Specifically, let $\mathsf{S} = \nu\mathsf{S}_3$. Then

$$(p,x) \in \mathsf{S}\,[m,p] \quad\iff x \sqsupseteq_{\text{SINK}} m$$

Proof: The proof that

$$\exists n, \pi_0 . m \notin \pi_0 \;\wedge\; x \to_G^{\pi_0} n \;\wedge\; \neg\, n \to_G^{*} m \quad\iff\quad \neg\, x \sqsupseteq_{\text{SINK}} m$$

is just as the proof that $\Pi_{\mathrm{SINK}}\,[x,m]$ characterizes $x \sqsupseteq_{\mathrm{SINK}} m$ in Observation B.2.1. But then the theorem follows immediately from Lemma B.3.2 and Lemma B.3.1. □

Remark B.3.2. A machine checked proof of Theorem B.3.1 using the Isabelle/HOL proof assistant[NWP02] was prepared by Simon Bischof. Bischofs proof does not invoke the dual functional $\overline{\mathsf{S}_3}$, and directly invokes co-inductive proof principles (while my proof of Lemma B.3.2 is purely inductive).

Observation B.3.2. Let S be computed by Algorithm 22. Then for all relevant m, p:

$$(\nu \mathsf{S}_3)\,[m,p] = \overline{\mathsf{S}\,[m,p]}$$

C A Slicing Algorithm using C-Edges

I don't even know what I was running for —
I guess I just felt like it.

(J.D. Salinger — The Catcher in the Rye)

Whenever a CFG G has no non-trivial control-sink, the corresponding pseudo-forests $<_{\text{SINK}}$ are *proper* forests[1]. Then by Observation 6.7.4,

$$\rightarrow^{\text{G}}_{\text{ntiod}} = \emptyset$$

and I can obtain control slices for arbitrary M from $\rightarrow^{\text{G}}_{\text{nticd}}$. In the special case that there is only one exit node n_x, I have $\rightarrow^{\text{G}}_{\text{nticd}} = \rightarrow^{\text{G}}_{\text{cd}}$, and by an algorithm[SG95] based on *DJ-Graphs*, any backward-slice

$$\left(\rightarrow^{G}_{\text{nticd}}\right)^{*}(M) \tag{C.1}$$

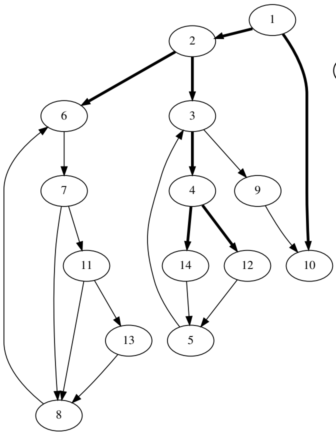
can be computed in linear time from $<_{\text{SINK}}$ without the construction of $\rightarrow^{\text{G}}_{\text{nticd}}$.

In this section, I will demonstrate that the technique from [SG95] to compute Equation C.1 can be generalized to *arbitrary* CFG with (possibly) *multiple* exit nodes $n_x, n'_x, \ldots$ and control-sinks.[2]

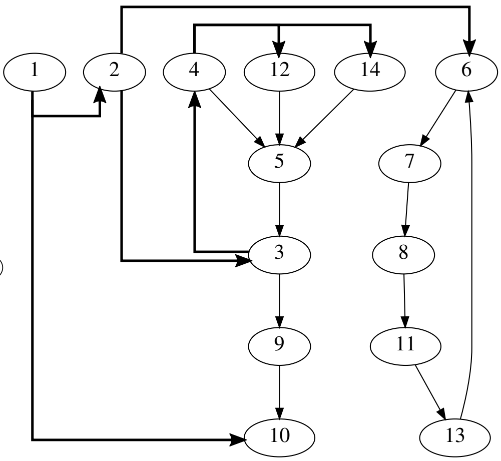
The original algorithm is presented with the application of placing ϕ-nodes in CFG with unique entry node n_e. It presupposes the *dominance*-tree, and computes the iterated dominance frontiers of sets M of variable definitions by considering *join*-edges $n \rightarrow_{\text{J}} m$, where

$$\begin{aligned} n \rightarrow_{\text{J}} m \quad &\Leftrightarrow \quad n \rightarrow_{G} m \;\wedge\; \neg n \sqsupset_{\text{DOM}} m \\ &\Leftrightarrow \quad n \rightarrow_{G} m \;\wedge\; n \neq \text{idom}\,(m) \end{aligned}$$

[1] i.e.: $<_{\text{SINK}}$ is free of cycles
[2] but i make no formal claim about the worst-case running time of this generalization



(a) A CFG, with $\rightarrow_C$ **bold**



(b) $<_{\text{SINK}} \cup \rightarrow_C$, with $\rightarrow_C$ **bold**

Figure C.1: Conditional Edges $\rightarrow_C$

In contrast, my application is backward-control-slicing in arbitrary CFG. I presuppose the pseudo forest $<_{\text{SINK}}$, and compute the iterated *post*dominance frontiers of sets M of slicing criteria by considering *conditional*-edges $n \rightarrow_C m$, where[3]

$$n \rightarrow_C m \quad \Leftrightarrow \quad n \rightarrow_G m \ \wedge \ m \notin \text{ipdom}_{\sqsupseteq_{\text{SINK}}}(n)$$

That is, my setting is obtained from the original by

1. as usual, considering CFG with edges $\rightarrow_G$ flipped, and then
2. generalizing to arbitrary CFG, by going from $\sqsupseteq_{\text{POST}}$ to $\sqsupseteq_{\text{SINK}}$.

[3] it is *not*, in general, the case that $n \rightarrow_C m \ \Leftrightarrow \ n \rightarrow_G m \ \wedge \ \neg m \sqsupseteq_{\text{SINK}} n$

I recall the example CFG from Figure 5.1a and the corresponding pseudo-forest $<_{\text{SINK}}$ in Figure C.1. Conditional edges $n \rightarrow_{\text{C}} m$ are **bold** in Figure C.1a, and have also been added to Figure C.1b.

In order to derive a generalized algorithm for the computation of Equation C.1, I will now present the appropriate generalizations of the relevant properties from [SG95]. I will require the following notation:

Definition C.0.1. The set of $\sqsupseteq_{\text{SINK}}$-*ancestors* of m is

$$\sqsupseteq_{\text{SINK}} \uparrow^{+} (m) \quad = \quad \{\, m'' \mid m'' \sqsupseteq_{\text{SINK}} m',\ m' \in \text{ipdom}_{\sqsupseteq_{\text{SINK}}}(m) \,\}$$

i.e.:, for a transitive, reflexive reduction $>_{\text{SINK}}$ of $\sqsupseteq_{\text{SINK}}$

$$= \quad \{\, m'' \mid m <^{+}_{\text{SINK}} m'' \,\}$$

similarily:

$$\begin{aligned} \sqsupseteq_{\text{SINK}} \uparrow^{*} (m) \quad &= \quad \{\, m'' \mid m'' \sqsupseteq_{\text{SINK}} m \,\} \\ &= \quad \{\, m'' \mid m <^{*}_{\text{SINK}} m'' \,\} \end{aligned}$$

The following observation is to be understood as a generalization of the (key) Lemma 3.1 in [SG95]:

Observation C.0.1. Let G be any CFG, and $n \neq m$. Then

$$n \rightarrow_{\text{nticd}} m \quad \Longleftrightarrow \quad n \rightarrow_{\text{C}} m_0 \quad \text{and} \quad m \sqsupseteq_{\text{SINK}} m_0 \quad \text{and} \quad \neg\, m \sqsupseteq_{\text{SINK}} n$$

for some node m_0. With regard to any corresponding pseudo-forest $<_{\text{SINK}}$:

$$n \rightarrow_{\text{nticd}} m \quad \Longleftrightarrow \quad n \rightarrow_{\text{C}} m_0 <^{*}_{\text{SINK}} m \qquad \text{and} \quad \neg\, n <^{*}_{\text{SINK}} m$$

for some node m_0.

Up to the flipping of edges in G, the authors in [SG95] assume a unique exit node n_x, and hence can assume $<_{\text{SINK}}$ to be a proper tree, which allows them to presuppose for each node n it's *level* $\text{lvl}\,(n)$, defined as its distance from the root.

Lemma C.0.1 ([SG95], Lemma 3.1, after flipping edges in G)**.** Let G be a CFG with unique exit node n_x, and $n \neq m$. Then $n \rightarrow_{\text{cd}} \ m$ iff there exists a node m_0 in the sub-tree rooted in m with conditional edge $n \rightarrow_{\text{C}} m_0$ and $\text{lvl}\,(n) \leq \text{lvl}\,(m)$.

So in my generalization,

m_0 in the sub-tree rooted in m	became	$m_0 <^*_{\text{SINK}} m$
$\text{lvl}\,(n) \leq \text{lvl}\,(m)$	became	$\neg\ n <^*_{\text{SINK}} m$

By Observation C.0.1, I can compute

$$(\rightarrow_{\text{nticd}})^*\,(M)$$

by following $(\rightarrow_{\text{C}} \ \cup <_{\text{SINK}})$-paths backwards from nodes $m \in M$, disregarding reached nodes n in accord with the condition $\neg\ n <^*_{\text{SINK}} m$. In the special case of a unique exit node n_x, there is an efficient algorithm to do this because by keeping a (never increasing) *current tree level* l,

1. no candidate node n ever needs to be visited twice (even if there paths from n to multiple nodes $m, m', \ldots \in M$),
2. the algorithm only ever needs to consider nodes n such that $\text{lvl}\,(n) \leq l$, which — unlike $\neg\ n <^*_{\text{SINK}} m$ — is a $\mathcal{O}\,(1)$ check!

So in order to devise an algorithm for the general case, I need a generalized notion of the "pseudo-forest level" of nodes n that allows me to replace checks $\neg\ n <^*_{\text{SINK}} m$ by a comparison of pseudo-forest levels.

Recall that in pseudo-forests $<$, roots are either single nodes $\{r\}$, or $<$-cycles R. So the following definition generalizes the notion of tree-level:

Definition C.0.2. For any pseudo forest $<$ and any node n, let $R_n \ \subseteq N$ be the root of the pseudo-tree of n. Then I define

$$\text{lvl}^{<}\,(n) = \min_{\pi = n, \ldots, r \quad r \in R_n} |\pi| \quad \geq 1$$

I also define the *level of the $<$-successor of n*:

$$\mathrm{lvl}^{<}_{\mathrm{next}}(n) = \begin{cases} \mathrm{lvl}(n') & \text{if } n < n' \\ 0 & \text{if there is no such } n' \end{cases}$$

Note that for "most" nodes, $\mathrm{lvl}^{<}_{\mathrm{next}}(n) = \mathrm{lvl}^{<}(n) - 1$, i.e.:

$$\mathrm{lvl}^{<}_{\mathrm{next}}(n) \ < \ k \quad \Longleftrightarrow \quad \mathrm{lvl}^{<}(n) \leq k$$

In a backward-traversal of $(\rightarrow_C \ \cup <_{\text{SINK}})$, I obviously cannot "jump trees by" following $<_{\text{SINK}}$. Also, as a consequence of the following Observation C.0.2, whenever I reach a candidate node n by backwards-following a conditional edge $n \rightarrow_C m_0 <^{*}_{\text{SINK}} m$:

1. n cannot be in any proper $<_{\text{SINK}}$-cycle (and specifically: $\neg\, n \in R_m$)
2. if n is in a different pseudo tree than m_0 and m, then $\{n\}$ is a root, i.e.: $\neg\, n <^{+}_{\text{SINK}} m'$ for *any* node m', and $\mathrm{lvl}^{<_{\text{SINK}}}(n) = 1$.
3. if $n \neq m$ is in the same pseudo-tree as m_0 and m, then $\neg\, n <^{*}_{\text{SINK}} m$ iff $\mathrm{lvl}^{<_{\text{SINK}}}_{\mathrm{next}}(n) \ < \ \mathrm{lvl}^{<_{\text{SINK}}}(m)$.

But this just means that regardless whether the edge $n \rightarrow_C m_0$ jumps trees or not, I can simply test $\mathrm{lvl}^{<_{\text{SINK}}}_{\mathrm{next}}(n) \ < \ \mathrm{lvl}^{<_{\text{SINK}}}(m)$.

Observation C.0.2. Let G be any CFG with corresponding pseudforest $<_{\text{SINK}}$. Then whenever $n \rightarrow_C m$,

$$\begin{aligned} \sqsupseteq_{\text{SINK}} \uparrow^{*}(m) \quad &\supset \quad \sqsupseteq_{\text{SINK}} \uparrow^{+}(n) \\ \text{and} \qquad\qquad m \quad &\notin \quad \sqsupseteq_{\text{SINK}} \uparrow^{+}(n) \end{aligned}$$

Note that the inequation is strict. For n that do have *some* immediate $\sqsupseteq_{\text{SINK}}$-dominator, this can be understood to mean that conditional edges $n \rightarrow_C m$ "never advance toward the root of n", but remain in the same tree. If n does not have any immediate $\sqsupseteq_{\text{SINK}}$-dominator, Observation C.0.2 makes no restriction, and $n \rightarrow_C m$ may "jump between pseudo-trees".

In the example Figure C.2 on page 393, I show a two-tree pseudo-forest $<_{\text{SINK}}$ and highlight for some n the set M_n of nodes m that "are allowed" by Observation C.0.2.

In Algorithm 17, I write just lvl for $\text{lvl}^{<_{\text{SINK}}}$ and lvl_{next} for $\text{lvl}_{\text{next}}^{<_{\text{SINK}}}$. Unlike the original algorithm for the special case with unique exit node n_x, my general algorithm cannot proceed with a never-increasing global *current level*, because I will sometimes discover new trees n via an edge $n \rightarrow_C m_0$. Instead, I hold fast to a current node m — selected from the priority-queue Q — whose sub-tree I explore, noting edges $n \rightarrow_C m_0$ *en passant*. The check $\mathsf{m}_0' \notin \mathsf{Visited}$ ensures that I never explore a node more than once.

Observation C.0.3. Let G be any CFG, M any set of nodes, and N be computed by Algorithm 17. Then

$$N = (\rightarrow_{\text{nticd}})^* (M)$$

I derived Algorithm 17 by generalizing the algorithm from [SG95] for classical (post-)dominance, which is a nontermination-insensitive notion. But Algorithm 17 works just as well for nontermination sensitive postdominance:

Observation C.0.4. Let G be any CFG, M any set of nodes, and N be computed by Algorithm 17, with input $>_{\text{MAX}}$ instead of $>_{\text{SINK}}$. Then

$$N = (\rightarrow \text{ntscd})^* (M)$$

Closing this section, let me note that since control-sinks do not contribute to $(\rightarrow_{\text{nticd}})^*$, it was to be expected that the original algorithm could be generalized to pseudo-*trees*. What may not have been obvious is my generalization to (pseudo)-*forests*. I also fully expect that the algorithm does not only work for $\sqsupseteq_{\text{SINK}}$ and $\sqsupseteq_{\text{MAX}}$, but for a general class of "postdominance like" relations $\sqsupseteq$. It would be interesting

```
Input  : A set $M$ of nodes (the slice criteria)
Input  : A transitive reduction $>_{\text{SINK}}$ of $\sqsupseteq_{\text{SINK}}$
Input  : The conditional edges $\rightarrow_C$
Output: The set $(\rightarrow_{\text{nticd}})^* (M)$
$\mathsf{N} \leftarrow M$
$\mathsf{Q} \leftarrow M$
$\mathsf{Visited} \leftarrow \emptyset$
while $\mathsf{Q} \neq \emptyset$ do
    $\mathsf{m} \leftarrow \text{remove}(\mathsf{Q})$ s.t. $\text{lvl}(\mathsf{m}) = \max_{m \in Q} \text{lvl}(m)$
    if $\mathsf{m} \notin \mathsf{Visited}$ then
        $\mathsf{M_0} \leftarrow \{\mathsf{m}\}$
        while $\mathsf{M_0} \neq \emptyset$ do
            $\mathsf{m_0} \leftarrow \text{remove}(\mathsf{M_0})$
            for $\mathsf{n} \rightarrow_C \mathsf{m_0},\ \mathsf{n} \notin \mathsf{N},\ \text{lvl}_{\text{next}}(\mathsf{n}) < \text{lvl}(\mathsf{m})$ do
                $\mathsf{N} \leftarrow \mathsf{N} \cup \{\mathsf{n}\}$
                $\mathsf{Q} \leftarrow \mathsf{Q} \cup \{\mathsf{n}\}$
            end

            for $\mathsf{m'_0} <_{\text{SINK}} \mathsf{m_0},\ \mathsf{m'_0} \notin \mathsf{Visited}$ do
                $\mathsf{M_0} \leftarrow \mathsf{M_0} \cup \{\mathsf{m'_0}\}$
            end
            $\mathsf{Visited} \leftarrow \mathsf{Visited} \cup \{\mathsf{m_0}\}$
        end
    end
end
return $\mathsf{N}$
```

Algorithm 17: Computation of $(\rightarrow_{\text{nticd}})^* (M)$ via $\rightarrow_C$

to check if, for example, the conditions for *admiting an efficient* PDF *partitioning* (Definition 3.2.7 on page 25) are already sufficient.

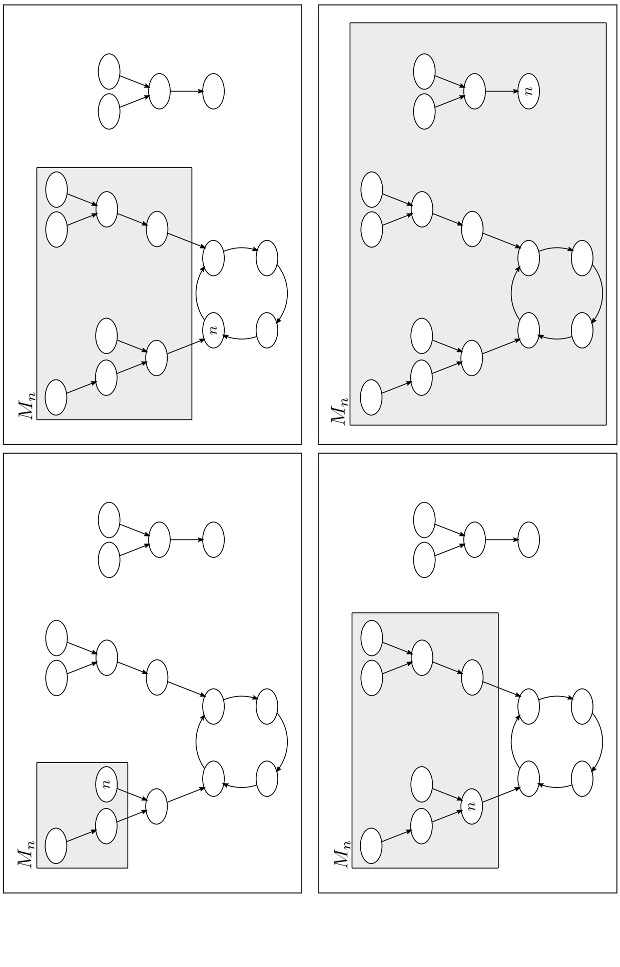


Figure C.2: The region M_n such that $n \to_C m$ is not prohibited by Observation C.0.2

D Algorithm Variants

An algorithm must be seen to be believed, and the best way to learn what an algorithm is all about is to try it.

(Donald Knuth — The Art of Computer Programming Vol. 1)

A slightly more efficient variant of Algorithm 3 is shown in Algorithm 18. Whenever it is known that the pseudo-forest $<$ is cycle-free, the checks $\mathsf{n}' \in \mathsf{ß}_\mathsf{n}$ and $\mathsf{m}' \in \mathsf{ß}_\mathsf{m}$ can be omitted.

Algorithm 19 is an algorithm for the computation of a pseudoforest $<_{\text{MAX}}$. It maintains a work-queue of nodes with fixed iteration order, instead of a work set.

Algorithm 20 and 21 compute a *cost demand*, i.e.: costs such that after adding them to the timing cost of edged towards nodes n_0, m_0, the returned node is a timing sensitive least common ancestor of n_0, m_0.

```
Input     : A pseudo-forest $<$, represented as a map
            $\mathsf{IMDOM} : N \hookrightarrow N$ s.t. $\mathsf{IMDOM}[n] = m$ iff $n < m$.
Input     : Nodes $\mathsf{m}_0$, $\mathsf{n}_0$
Output    : A least common ancestor of $\mathsf{n}_0, \mathsf{m}_0$, or $\perp$ if there is
            none.
begin
    return lca ( $\mathsf{n}_0$, $\mathsf{m}_0$)
end
Function lca ($\pi_\mathsf{n}$ , $\pi_\mathsf{m}$)
    Input      : A $<$-path $\pi_\mathsf{n} = \mathsf{n}_0, \ldots, \mathsf{n}$ ending in $\mathsf{n}$
    Input      : A $<$-path $\pi_\mathsf{m} = \mathsf{m}_0, \ldots, \mathsf{m}$ ending in $\mathsf{m}$
    if $\mathsf{m} \in \pi_\mathsf{n}$ then return $\mathsf{m}$
    switch IMDOM[ n ] do
        case $\perp$ do return lin[$\pi_\mathsf{n}$] ($\pi_\mathsf{m}$)
        case n' do
            if n' $\in \pi_\mathsf{n}$ then
                return lin[$\pi_\mathsf{n}$] ($\pi_\mathsf{m}$)
            end
            return lca($\pi_\mathsf{m}$, $\pi_\mathsf{n}$ n')
        end
    end
end
Function lin[$\pi_\mathsf{n}$] ($\pi_\mathsf{m}$)
    Input      : A $<$-path $\pi_\mathsf{m} = \mathsf{m}_0, \ldots, \mathsf{m}$ ending in $\mathsf{m}$
    Implicit   : A $<$-path $\pi_\mathsf{n} = \mathsf{n}_0, \ldots, \mathsf{n}$ ending in $\mathsf{n}$
    switch IMDOM[m] do
        case $\perp$ do return $\perp$
        case m' do
            if m' $\in \pi_\mathsf{n}$ then return m'
            if m' $\in \pi_\mathsf{m}$ then return $\perp$
            return lin($\pi_\mathsf{m}$m')
        end
    end
end
```

Algorithm 18: A *least common ancestor* algorithm variant of Algorithm 3

```
Input   : A CFG $G$
Data: A pseudo-forest $<$ represented as a map $\mathsf{IMDOM} : N \hookrightarrow N$ s.t.
      $\mathsf{IMDOM}[n] = m$ iff $n < m$
Output: A transitive reduction $<_{\text{MAX}}$ of $\sqsupseteq_{\text{MAX}}$
begin
    for $x \in N$, $\{z \mid x \rightarrow_G z\} = \{z\}$, $z \neq x$ do
        $\mathsf{IMDOM}[x] \leftarrow z$
    end
    MAXIMAL$_{\text{up}}$
    return $\mathsf{IMDOM}$
end
Procedure MAXIMAL$_{\text{up}}$
    $\mathsf{workqueue} \leftarrow \text{COND}_G$
    $\mathsf{oldest} \leftarrow \bot$
    while $\mathsf{workqueue} \neq \emptyset$ do
        $x \leftarrow \text{removeFront}(\mathsf{workqueue})$
        assert $\mathsf{IMDOM}[x] = \bot$
        if $\mathsf{oldest} = x$ then
            return
        end
        if $\mathsf{oldest} = \bot$ then
            $\mathsf{oldest} \leftarrow x$
        end
        $a \leftarrow \mathsf{lca}(\{ y \mid x \rightarrow_G y \})$
        $z \leftarrow \begin{cases} \bot & \text{if } a = \bot \lor a = x \\ a & \text{otherwise} \end{cases}$
        if $z \neq \bot$ then
            $\mathsf{IMDOM}[x] \leftarrow z$
            $\mathsf{oldest} \leftarrow \bot$
        end
        else
            $\text{pushBack}(\mathsf{workqueue}, x)$
        end
    end
end
```

Algorithm 19: An efficient algorithm for the computation of some $<_{\text{MAX}}$.

```
Input       : A $\mathbb{N}$ labeled pseudo-forest $<$, represented as a map
              $\mathsf{IDOM} : N \hookrightarrow N \times \mathbb{N}$ s.t. $\mathsf{IDOM}[n] = (m,k)$ iff $n <^k m$
Input       : Numbers $\mathsf{k}_0^\mathsf{n}$, $\mathsf{k}_0^\mathsf{m} \in \mathbb{N}$ and nodes $\mathsf{n}_0$, $\mathsf{m}_0$ such that *some*
              $<_{\text{MAX}}$-least common ancestor exists
Output      : A triple $(\mathsf{a},\ \mathsf{k},\ \Delta\mathsf{C})$ such that $\Delta\mathsf{C}$ is a map with
              $(\mathsf{a},\mathsf{k}) = \text{lca}_<\left((\mathsf{n}_0, \mathsf{k}_0^\mathsf{n} + \Delta\mathsf{C}[\mathsf{n}_0]), (\mathsf{m}_0, \mathsf{k}_0^\mathsf{m} + \Delta\mathsf{C}[\mathsf{m}_0])\right)$, or
              $\perp$ if no such triple exists
begin
    return $\mathsf{lca}\left((\mathsf{n}_0, \mathsf{k}_0^\mathsf{n}, [\mathsf{n}_0 \mapsto \mathsf{k}_0^\mathsf{n}]),\ (\mathsf{m}_0, \mathsf{k}_0^\mathsf{m}, [\mathsf{m}_0 \mapsto \mathsf{k}_0^\mathsf{m}])\right)$
end
Function $\mathsf{lca}(\pi_\mathsf{n},\ \pi_\mathsf{m})$
    Input       : A cycle-free $<$-path $\pi_\mathsf{n} = \mathsf{n}_0, \ldots, \mathsf{n}$ ending in $\mathsf{n}$,
                  represented by a tuple $(\mathsf{n}, \mathsf{k}^\mathsf{n}, \mathsf{KS}_\mathsf{n})$ where $\mathsf{KS}_\mathsf{n}$ is a
                  map on the nodes $n$ appearing in $\pi_\mathsf{n}$ s.t.
                  $\mathsf{k}^\mathsf{n} = \mathsf{KS}_\mathsf{n}[\mathsf{n}]$ and for any such $n$
                    $\mathsf{KS}_\mathsf{n}[n] = \mathsf{k}_0^\mathsf{n} + \sum_i k_i$ for $\mathsf{n}_0 <^{k_1} \ldots <^{k_c} n$ in $\pi_\mathsf{n}$
    Input       : A $<$-path $\pi_\mathsf{m} = \mathsf{m}_0, \ldots, \mathsf{m}$ likewise
    if $\mathsf{m} \in \pi_\mathsf{n}$ then let $\Delta C = |\mathsf{k}^\mathsf{m} - \mathsf{KS}_\mathsf{n}[\mathsf{m}]|$ in
        if $\mathsf{k}^\mathsf{m} = \mathsf{KS}_\mathsf{n}[\mathsf{m}]$ then return $(\mathsf{m}, [\mathsf{n}_0 \mapsto 0, \mathsf{m}_0 \mapsto 0])$
        if $\mathsf{k}^\mathsf{m} < \mathsf{KS}_\mathsf{n}[\mathsf{m}]$ then return $(\mathsf{m}, [\mathsf{n}_0 \mapsto 0, \mathsf{m}_0 \mapsto \Delta C])$
        if $\mathsf{k}^\mathsf{m} > \mathsf{KS}_\mathsf{n}[\mathsf{m}]$ then return $(\mathsf{m}, [\mathsf{n}_0 \mapsto \Delta C, \mathsf{m}_0 \mapsto 0])$
    end
    switch $\mathsf{IDOM}[\mathsf{n}]$ do
        case $\perp$ do return $\mathsf{lin}[\pi_\mathsf{n}](\pi_\mathsf{m})$
        case $(\mathsf{n'}, \mathsf{k})$ do
            if $\mathsf{n'} \in \pi_\mathsf{n}$ then return $\mathsf{lin}[\pi_\mathsf{n}](\pi_\mathsf{m})$
            return $\mathsf{lca}(\pi_\mathsf{m},\ \pi_{\mathsf{n'}})$ where $\pi_{\mathsf{n'}} = (\mathsf{n'}, \mathsf{k}^\mathsf{n} + \mathsf{k}, \mathsf{KS}_\mathsf{n}[\mathsf{n'} \mapsto \mathsf{k}^\mathsf{n} + \mathsf{k}])$
        end
    end
end
```

Algorithm 20: A *least common ancestor* algorithm that computes a timing cost demand, continued in Algorithm 21

```
Function lin[$\pi_n$] ($\pi_m$)
    Input    : A $<$-path $\pi_m = m_0, \ldots, m$ ending in m, represented
               as in Algorithm 20
    Implicit : A $<$-path $\pi_n = n_0, \ldots, n$ ending in n, likewise
    switch IDOM[m] do
        case $\perp$ do return $\perp$
        case (m', k) do let $k^{m'} = k^m + k$ in
            if $m' \in \pi_n$ then let $\Delta C = |k^{m'} - KS_n[m']|$ in
                if $k^{m'} = KS_n[m']$ then return $(m', [n_0 \mapsto 0, m_0 \mapsto 0])$
                if $k^{m'} < KS_n[m']$ then return $(m', [n_0 \mapsto 0, m_0 \mapsto \Delta C])$
                if $k^{m'} > KS_n[m']$ then return $(m', [n_0 \mapsto \Delta C, m_0 \mapsto 0])$
            end
            if $m' \in \pi_m$ then return $\perp$
            return $\mathsf{lin}(\pi_{m'})$ where $\pi_{m'} = (m', k^{m'}, KS_m[m' \mapsto k^{m'}])$
        end
    end
end
```

Algorithm 21: A *least common ancestor* algorithm that computes a timing cost demand (continued from Algorithm 20)

D.1 Another Algorithm for $\rightarrow_{\text{nticd}}$

Owing to Lemma B.3.1, I can give another work-set algorithm (Algorithm 22) for the computation of $\rightarrow_{\text{nticd}}$. I use a variant $\text{prev}^{\rightarrow}_{\text{COND}}$ of $\text{prev}_{\text{COND}}$ that — along with predecessor-conditional nodes n of some node p — also gives me the node(s) x s.t. $n \rightarrow_G x \rightarrow^*_G p$.

```
**Input**  : A CFG $G = (N, E)$
**Input**  : Maps $\overline{\text{next}_{\text{COND}}}$ and $\text{prev}^{\rightarrow}_{\text{COND}}$
**Output** : A map $\mathsf{NTICD}$ such that $\mathsf{NTICD}[n] = \{m \mid n \rightarrow_{\text{nticd}} m\}$
**Data:** A map $\mathsf{S}$ from pairs of nodes to sets of nodes
**Data:** A workset $\mathsf{workset} \subseteq N \times \text{COND}_G \times N$
**begin**
    **for** $p \in \text{COND}_G,\ p \rightarrow_G x,\ \neg\, x \rightarrow_G m$ **do**
        $\mathsf{S}[m,p] \leftarrow \mathsf{S}[m,p] \cup \{(p,x)\}$
    **end**
    **for** $p \in \text{COND}_G,\ m \in N,\ |\mathsf{S}[m,p]| \neq 0,\ (n,x) \in \text{prev}^{\rightarrow}_{\text{COND}}[p]$ **do**
        $\mathsf{workset} \leftarrow \mathsf{workset} \cup \{(m,n,x)\}$
    **end**
    **while** $\mathsf{workset} \neq \emptyset$ **do**
        $(m,p,x) \leftarrow remove(\mathsf{workset})$
        **if** $\neg m \in \overline{\text{next}_{\text{COND}}}[x]$ **and** $\neg\,(p,x) \in \mathsf{S}[m,p]$ **then**
            **if** $|\mathsf{S}[m,p]| = 0$ **then**
                $\mathsf{workset} \leftarrow \{\,(m,n,x') \mid (n,x') \in \text{prev}^{\rightarrow}_{\text{COND}}[p]\,\}$
            **end**
            $\mathsf{S}[m,p] \leftarrow \mathsf{S}[m,p] \cup \{(p,x)\}$
        **end**

    **end**

    **for** $n \in N,\ m \in \text{COND}_G$ **do**
        **if** $0 < |\mathsf{S}[m,n]| < |\{x \mid n \rightarrow_G x\}|$ **then**
            $\rightarrow_{\text{nticd}}[m] \leftarrow \rightarrow_{\text{nticd}}[m] \cup \{n\}$
        **end**
    **end**
**end**
```

Algorithm 22: A *least* fixed point algorithm for $\rightarrow_{\text{nticd}}$.

D.2 Efficient $\mathrm{lca}_<$ via Postorder Numbers

In Algorithm 5 and Algorithm 6, the intersection of the postdominance sets of all successors of x of a conditional node p, i.e. the computation of the set of nodes m s.t. $m \sqsupseteq p$ via the rule (see: Theorem 5.1.2)

$$\frac{\forall p \to_G x.\ m \sqsupseteq x \qquad p \to_G^* m}{m \sqsupseteq p}\,\mathsf{D}^{\mathrm{suc}}$$

was replaced by it's abstraction in $<$, i.e. by the computation of

$$\mathrm{lca}_< (\{\, x \mid p \to_G x \,\})$$

In [CHK01], the algorithm for the computation of dominance in graphs with unique entry node n_e similarly computes the set of nodes m s.t. $m \sqsupseteq_{\mathsf{DOM}} p$ (by considering all predecessors x of join-nodes p), but employs one additional abstraction:

There, the least common ancestor $\mathrm{lca}_< (\{\, x \mid x \to_G p \,\})$ is also computed by following paths in $<$ and terminating once the path $\pi_n = n_0, \ldots, n$ from one predecessor n_0 joins the path π_m from another predecessor m_0 (as is done by Algorithm 3), but the check $n \in \pi_m$ is replaced by an (arithmetic) comparison of the nodes' postorder numbers.

In this subsection, i will demonstrate how this very technique can be adapted to the computation of $<_{\mathrm{SINK}}$.

First, i observe that the phase $\mathrm{SINK}_{\mathrm{up}}$ in Algorithm 6 can be replaced by a depth-first traversal in the reversed graph G^{-1}. Remember that i need to compute an approximation $>_0$ of $>_{\mathrm{SINK}}$, i.e. a pseudo-forest $>_0$ such that

$$>_0^* \quad \supseteq \quad \sqsupseteq_{\mathrm{SINK}}$$

Observation D.2.1. Let $G = (N, E)$ be any CFG, $S_1, \ldots, S_n$ its sinks, and $S = \bigcup S_i$. Choose a distinct node s_i for each S_i. Also choose for each S_i a fixed ordering $n_1, \ldots, n_k$ of each S_i.

Let $F \subseteq N \times N$ be the depth-first forest obtained from a search in G^{-1}, iteratively starting from the fixed nodes s_i, and let

$$\begin{aligned} <_0 \quad = \quad & \bigcup_i \{ \left(n_j, n_{(j+1 \mod k)} \right) \mid n_j \in S_i \} \\ & \cup \{ (m, n) \mid (n, m) \in F,\ m \notin S \} \end{aligned}$$

Then

$$>_0^* \quad \supseteq \quad \sqsupseteq_{\text{SINK}}$$

With regard to the precision of this $>_0$ compared with hat computed by phase SINK$_{\text{up}}$ in Algorithm 6, experiments with randomly generated graphs suggest that the result of SINK$_{\text{up}}$ is for *most* graphs more precise than that of the depth first search (if measured by the cardinality of $>_0^*$) and leads to quicker termination of phase SINK$_{\text{down}}$; but this is not true for every graph G.

Next i observe that, given a post-order numbering $^\# : N \to \mathbb{N}^+$ of the depth-first search as obtained in Observation D.2.1, the lca$_<$ (n, m) of any approximation $<$ of $\sqsupseteq_{\text{SINK}}$ with which that numbering is *compatible* can be computed using *arithmetic* comparisons:

Definition D.2.1. Let $<$ be a pseudo-forest with cycles S_i, and $S = \bigcup_i S_i$.

A numbering[1] $^\# : N \to \mathbb{N}$ is called *compatible* with $<$ if

1. for $n \notin S$ and $n < m$: $\quad n^\# < m^\#$
2. for $n \in S_i$ and $n < m$: $\quad m = s_{j+1 \mod k}$

[1] i.e.: an injective map

given that $n = s_j$ and $S_i = s_1, \ldots, s_k$ ordered by the numbering $^{\#}$.

Observation D.2.2. In the situation of Definition D.2.1, and given $n_0, m_0 \in N$, Algorithm 23 computes a number $a^{\#}$ such that $a \in \text{lca}_< (n, m)$, or 0 if $\text{lca}_< (n, m) = \emptyset$.

Proof (Sketch): Upon terminating with $a^{\#} \neq 0$, the algorithm has followed two strictly ascending[2] sequences $\pi_n^{\#} = n_0^{\#}, \ldots, n^{\#}$ and $\pi_m^{\#} = m_0^{\#}, \ldots, m^{\#}$ such that $n^{\#} = m^{\#}$ and $n_0 < \ldots < n$ and $m_0 < \ldots < m$, i.e.: $n_0 <^* a \wedge m_0 <^* a$. Also, since π_n, π_m are disjunct up to $m = n$, $a < a'$ for any other common ancestor of n_0, m_0. If, on the other hand, the algorithm terminates with 0, then — up to symmetry — either

- $n^{\#} = 0$, $m^{\#} \neq 0$, i.e.: $\pi_n^{\#} = n_0^{\#}, \ldots, n'^{\#}, 0$ and $\pi_m^{\#} = m_0^{\#}, \ldots, m^{\#}$ with $\pi_n \cap \pi_m = \emptyset$ and $n' < x$ for *no* $x \in N$, or
- $\pi_n^{\#} = n_0^{\#}, \ldots, n^{\#}$ and $\pi_m^{\#} = m_0^{\#}, \ldots, m^{\#}$ with $m < m'$, $m'^{\#} \leq m^{\#}$ and $n^{\#} > m^{\#}$. But because $^{\#}$ is compatible, $m^{\#} = \max_{m_0 <^* x} x^{\#} = s_k^{\#}$, with $s_k \in S_i$. Since $n^{\#} > m^{\#}$, there is no node $s \in S_i$ s.t. $n_0 <^* s$, and with $\pi_n \cap \pi_m = \emptyset$, it follows that $\text{lca}_< (n_0, m_0) = \emptyset$.

Ordering each sinks nodes by $^{\#}$, any approximation $<_0$ obtained via Observation D.2.1 is obviously compatible with the corresponding post-order numbering $^{\#}$. If i am to use Algorithm 23 in an algorithm phase similiar to $\text{SINK}_{\text{down}}$, compatibility has to be upheld — but this is simply to the fact that initially, if $n <_0^* m$ and $n \notin S$, not only $n^{\#} < m^{\#}$ (compatibility), but also $n \to_G m$. Since for $a = \text{lca}_< (\{ x \mid n \to_G x \})$ as computed by Algorithm 23 i have $\forall x. x^{\#} \leq a^{\#}$, i retain $n^{\#} < a^{\#}$ whenever i update $<$ at n with such a least common ancestor of the successors of n.

Empirical Observation D.2.1. Let G be any CFG. Then Algorithm 24 terminates with a result $<_{\text{SINK}}$ s.t. $>_{\text{SINK}}$ is a transitive reduction of $\sqsupseteq_{\text{SINK}}$.

[2] due to compatibility

```
Implicit     : A pseudo-forest $<$ with cycles $S_i$, represented as a map
               $\mathsf{ISDOM}^\# : \mathbb{N} \to \mathbb{N}$ s.t. $\mathsf{ISDOM}^\# [n^\#] = m^\#$ iff $n < m$,
               and $\mathsf{ISDOM}^\# [n^\#] = 0$ if there is no such $m$, for a
               postorder numbering $^\# : N \to \mathbb{N}^+$ compatible with $<$.
Input        : Postorder Numbers Nodes $\mathsf{m}_0^\#$, $\mathsf{n}_0^\#$ of Nodes $\mathsf{m}_0$, $\mathsf{n}_0$
Output       : The Postorder Number of a least common ancestor of
               $\mathsf{n}_0, \mathsf{m}_0$, or 0 if there is none.
Function $\mathsf{lca}^\# (\mathsf{n}^\#, \mathsf{m}^\#)$
    Input        : A Postorder Number $\mathsf{n}^\#$ such that $\pi_\mathsf{n} = \mathsf{n}_0, \ldots, \mathsf{n}$
                   is a $<$-path ending in $\mathsf{n}$
    Input        : A Postorder Number $\mathsf{m}^\#$ such that $\pi_\mathsf{m} = \mathsf{m}_0, \ldots, \mathsf{m}$
                   is a $<$-path ending in $\mathsf{m}$
    if $\mathsf{m}^\# = 0 \lor \mathsf{n}^\# = 0$ then return 0
    if $\mathsf{n}^\# > \mathsf{m}^\#$ then
        $\mathsf{m}'^\# \leftarrow \mathsf{ISDOM}[\mathsf{m}^\#]$
        if $\mathsf{m}'^\# \leq \mathsf{m}^\#$ then
            return 0                                             $(\mathsf{m}' \in \pi_\mathsf{m})$
        end
        else return $\mathsf{lca}^\# (\mathsf{n}^\#, \mathsf{m}'^\#)$
    end
    if $\mathsf{n}^\# = \mathsf{m}^\#$ then return $\mathsf{n}^\#$
    if $\mathsf{n}^\# < \mathsf{m}^\#$ then
        $\mathsf{n}'^\# \leftarrow \mathsf{ISDOM}[\mathsf{n}^\#]$
        if $\mathsf{n}'^\# \leq \mathsf{n}^\#$ then
            return 0                                             $(\mathsf{n}' \in \pi_\mathsf{n})$
        end
        else return $\mathsf{lca}^\# (\mathsf{n}'^\#, \mathsf{m}^\#)$
    end
end
```

Algorithm 23: A *least common ancestor* algorithm for nodes represented by their postorder number

```
Input  : A CFG $G$
Data   : A pseudo-forest $<$, represented as a map $\mathsf{ISDOM}^\# : \mathbb{N} \to \mathbb{N}$
         s.t. $\mathsf{ISDOM}^\# \left[n^\#\right] = m^\#$ iff $n < m$, and $\mathsf{ISDOM}^\# \left[n^\#\right] = 0$ if
         there is no such $m$
Output: A transitive reduction $<_{\text{SINK}}$ of $\sqsupset_{\text{SINK}}$
begin
    $\{S_1, \ldots, S_n\} \leftarrow \{S_k \mid S_k \in \mathsf{scc}(G), \neg \exists s \to_G n.\ s \in S_k \wedge n \notin S_k\}$
    $S \leftarrow \bigcup S_i$
    for $1 \leq i \leq n$ do
        $s_i \leftarrow$ any node in $S_i$
    end
    let $<_0$ be obtained by a depth first search in $G^{-1}$ (see:
     Observation D.2.1), compatible with post-order numbering $^\#$
    for $n <_0 m$ do
        $\mathsf{ISDOM}^\# \left[n^\#\right] \leftarrow m^\#$
    end
    $\mathrm{SINK}^\#_{\mathrm{down}}$
    return $\mathsf{ISDOM}$
end
Procedure $\mathrm{SINK}^\#_{\mathrm{down}}$
    repeat
        $\mathsf{changed} \leftarrow$ false
        for $x \in \mathrm{COND}$ do
            $a^\# \leftarrow \mathsf{lca}^\# \left(\{ y^\# \mid x \to_G y \}\right)$
            $z^\# \leftarrow \begin{cases} 0 & \text{if } a^\# = 0 \\ s_i^\# & \text{if } a \in S_i \\ a^\# & \text{otherwise} \end{cases}$
            if $z^\# \neq \mathsf{ISDOM}^\# \left[x^\#\right]$ then
                $\mathsf{ISDOM}^\# \left[x^\#\right] \leftarrow z^\#$
                $\mathsf{changed} \leftarrow$ true
            end
        end
    until $\neg\mathsf{changed}$
end
```

Algorithm 24: An Algorithm for the computation of some $<_{\text{SINK}}$.

The fixed point iteration in Algorithm 24 is the most naive implementation possible. Several worklist based optimizations are possible. Again, one can also make use of the fact that once $\mathsf{ISDOM}^{\#}\left[x^{\#}\right] = 0$, x no longer needs to be considered. As is, Algorithm 24 can be read as a strict generalization[3] of the algorithm in [CHK01].

[3] after the transition from G to G^{-1}

E Generalizations for CFG with Timing Cost Model

This is a real frickin' embarrassment.

(Scout — Teamfortress 2)

In Subsection 9.4.3, I described an algorithm (Algorithm 11) for the computation of the timing sensitive postdominance frontier $\mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$ for CFG under the (implicit) uniform timing cost model $\mathbb{1}$ which assigns a duration of 1 unit of time to each edge $n \rightarrow_G m$.

The algorithm was derived from Observation 9.2.3 on page 176 for

$$\sqsupseteq \;=\; \sqsupseteq_{\mathrm{TIME[FIRST]}} \;=\; \sqsupseteq^{\mathbb{1}}_{\mathrm{TIME[FIRST]}}$$

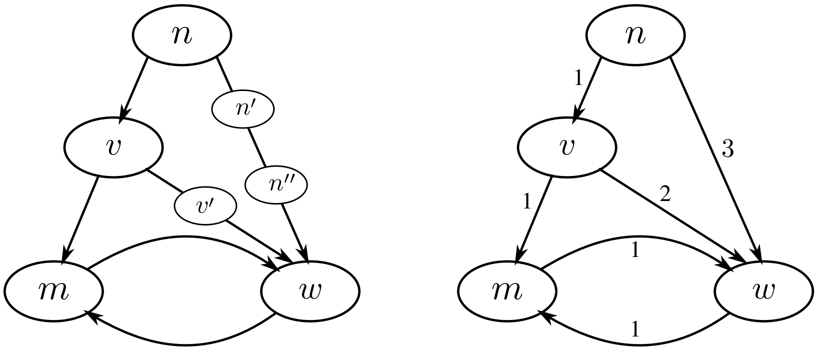
via a corresponding least fixed point characterization in Figure 9.5 on page 178.

I did not justify why it is enough to take the *least* fixed of rule system in Figure 9.5. In fact, Observation 9.2.3 merely characterizes, for each node x, $\mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}(x)$ in terms of some sets $\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}(x,z)$, and hence ultimately: in terms of $\mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}(z)$ of some nodes z. In other words: it is — in general — merely a *mutually recursive* system of equations, and Algorithm 11 merely states that $\mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$ is *some* solution of this system, but — a priori — not necessarily the least.

Consider the CFG in Figure E.1a under the default timing cost $\mathbb{1}$.

For $\sqsupseteq = \sqsupseteq_{\mathrm{TIME[FIRST]}}$, I have

$$\mathrm{ipdom}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}(m) \;=\; \mathrm{ipdom}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}(w) \;=\; \{m, w\}$$



(a) A CFG G (b) A CFG G_C with timing cost C

Figure E.1: The need for a modification of Algorithm 11.

and so from Observation 9.2.3 i merely obtain

$$\begin{aligned}
\mathrm{PDF}_{\sqsupseteq}(m) &= \underbrace{\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(m)}_{=\ \{v\}} \cup \ldots \cup \underbrace{\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(m,m) \cap \sqsubseteq'_w}_{\subseteq\ \ \mathrm{PDF}_{\sqsupseteq}(m)} \\
\text{and}\quad \mathrm{PDF}_{\sqsupseteq}(w) &= \underbrace{\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(w)}_{=\ \emptyset} \cup \ldots \cup \underbrace{\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(m,m) \cap \sqsubseteq'_w}_{\subseteq\ \ \mathrm{PDF}_{\sqsupseteq}(w)}
\end{aligned}$$

Note that the inequation $\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(m,m) \cap \sqsubseteq'_w \subseteq\ \mathrm{PDF}_{\sqsupseteq}(m)$ is already due to the definition

$$\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(m,m) \ = \ \{\, y \in \mathrm{PDF}_{\sqsupseteq}(m) \mid \neg\, m \text{ 1-}{\sqsupseteq}\, y \,\}$$

of $\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(m,m)$.

At the same time I observe — by manual inspection of G — that m is timing sensitively dependent on n (i.e.: $n \in \mathrm{PDF}_{\sqsupseteq}(m)$). Since in the equation for $\mathrm{PDF}_{\sqsupseteq}(m)$, n does not appear left from the ellipsis $\ldots$, and the term on the right of the ellipsis is not "productive" (it is, after all, a subset of $\mathrm{PDF}_{\sqsupseteq}(m)$) in a least fixed point computation, the node

n must "reach" $\mathrm{PDF}_{\sqsupseteq}(m)$ via some term *in* the ellipsis. And indeed it does: realizing that $n \in \mathrm{PDF}_{\sqsupseteq}(n'')$, and that $m \in \mathrm{ipdom}_{\sqsupseteq}(n'')$, but $\neg\, n'' \in \mathrm{ipdom}_{\sqsupseteq}(m)$, the node n reaches $\mathrm{PDF}_{\sqsupseteq}(m)$ via

$$\mathrm{PDF}_{\sqsupseteq}(m) \quad = \quad \ldots \cup \mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(n'', m) \cup \ldots$$

In summary, everything is fine for $\mathrm{PDF}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}} = \mathrm{PDF}^{\mathbb{1}}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$. Even though the equation for $\mathrm{PDF}_{\sqsupseteq}(m)$ is (syntactically) not fully productive whenever $m \in \mathrm{ipdom}_{\sqsupseteq}(m)$ (i.e.: when m is in a non-trivial $<_{\mathrm{TIME}}$ cycle M), nevertheless all $y \in \mathrm{PDF}_{\sqsupseteq}(m)$ are provided either by $\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(m')$ for some $m' \in M$, or by $\mathrm{PDF}_{\sqsupseteq}(n')$ for some node $n' \notin M$ such that $m \in \mathrm{ipdom}_{\sqsupseteq}(n')$.

In contrast, things are *not* generally fine for $\mathrm{PDF}^{C}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$ for *arbitrary* timing cost model C. Both Observation 9.2.3, and the simplifications Observation 9.2.4 and Observation 9.2.5 of $\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}$ and $\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}$ *do* hold even for $\sqsupseteq\, = \sqsupseteq^{C}_{\mathrm{TIME[FIRST]}}$ and arbitrary G, C. But for $\mathrm{PDF}^{C}_{\sqsupseteq_{\mathrm{TIME[FIRST]}}}$, it is no longer enough to simply take the least fixed point of this system of equations.

Consider the CFG and the timing cost model C shown in Figure E.1b. Up to *intermediate* nodes v', n', n'', this is essentially the same CFG as before.

Again, I have

$$\mathrm{ipdom}_{\sqsupseteq^{C}_{\mathrm{TIME[FIRST]}}}(m) \;=\; \mathrm{ipdom}_{\sqsupseteq^{C}_{\mathrm{TIME[FIRST]}}}(w) \;=\; \{m, w\}$$

and so from Observation 9.2.3 I obtain gain, for $\sqsupseteq\, = \sqsupseteq^{C}_{\mathrm{TIME[FIRST]}}$,

$$\begin{aligned} \mathrm{PDF}_{\sqsupseteq}(m) \quad &= \quad \underbrace{\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(m)}_{=\ \{v\}} \cup \ldots \cup \underbrace{\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(m, m) \cap \sqsubseteq'_{w}}_{\subseteq\ \ \mathrm{PDF}_{\sqsupseteq}(m)} \\ \text{and} \quad \mathrm{PDF}_{\sqsupseteq}(w) \quad &= \quad \underbrace{\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(w)}_{=\ \emptyset} \cup \ldots \cup \underbrace{\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(m, m) \cap \sqsubseteq'_{w}}_{\subseteq\ \ \mathrm{PDF}_{\sqsupseteq}(w)} \end{aligned}$$

I observe — again by manual inspection, this time of CFG G_C under timing cost model C – that m is timing sensitively dependent on n under timing cost model C (i.e.: $n \in \mathrm{PDF}_{\sqsupseteq}(m)$).

But can I obtain $n \in \mathrm{PDF}_{\sqsupseteq}(m)$ by a least fixed point iteration? In other words, can I obtain $n \in \mathrm{PDF}_{\sqsupseteq}(m)$ purely from the center ellipsis … for m? First, I must observe that

$$m \notin \mathrm{ipdom}_{\sqsupseteq^{C}_{\mathrm{TIME[FIRST]}}}(v) \;=\; \mathrm{ipdom}_{\sqsupseteq^{C}_{\mathrm{TIME[FIRST]}}}(n) \;=\; \{w\}$$

In other words: now there exists *no* node n'' outside of $M = \{m, w\}$ such that $m \in \mathrm{ipdom}_{\sqsupseteq^{C}_{\mathrm{TIME[FIRST]}}}(n'')$. Also, $w \in M$ is *not* timing sensitively control dependent on n under cost model C: $\neg\, n \in \mathrm{PDF}_{\sqsupseteq}(w)$.

The full equation for m thus reads: $\mathrm{PDF}_{\sqsupseteq}(m) \;=$

$$\underbrace{\mathrm{PDF}^{\mathrm{local}}_{\sqsupseteq}(m)}_{=\;\{v\}} \;\cup\; \underbrace{\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(w,m) \cap \sqsubseteq'_m}_{\subseteq\;\mathrm{PDF}_{\sqsupseteq}(w)} \;\cup\; \underbrace{\mathrm{PDF}^{\mathrm{up}}_{\sqsupseteq}(m,m) \cap \sqsubseteq'_m}_{\subseteq\;\mathrm{PDF}_{\sqsupseteq}(m)}$$

which is *not* productive for $n \in \mathrm{PDF}_{\sqsupseteq}(m)$, so I can *not* obtain $\mathrm{PDF}_{\sqsupseteq}(m)$ as a least fixed point of the implied functional (nor as the least fixed point from the simplified rule system Figure 9.5).

I did *not* investigate whether $\mathrm{PDF}_{\sqsupseteq}$ can be obtained as the greatest fixed point of Figure 9.5 or some related system. Instead I observe that nodes n missing in the least fixed point computation for $\mathrm{PDF}_{\sqsupseteq}(m)$ must always be nodes at the *border* N_M of the $<^{C}_{\mathrm{TIME}}$-cycle M that m is part of. This means that I *can* compute $\mathrm{PDF}_{\sqsupseteq}$ as a least fixed point, if I augment the rule system Figure 9.5 with the additional rule

$$\frac{\neg\, x \in \mathrm{ipdom}_{\sqsupseteq}(y) \qquad x \in M \qquad M \in \mathbb{M} \qquad y \in N_M \qquad \begin{matrix} y \to_G y' \\ x \sqsupseteq y' \end{matrix}}{y \in \mathrm{PDF}_{\sqsupseteq}(x)}\;\mathrm{PDF}$$

where $\mathbb{M}$ denotes the set of non-trivial $<^{C}_{\mathrm{TIME}}$-cycles. The rule is directly derived from the *definition* of $\mathrm{PDF}_{\sqsupseteq}$, but restricted to a *small* *subset* of pairs x, y of nodes. Note that since $x \in M$, the condition $x \sqsupseteq y'$ can be replaced by

$$y' \in M \quad \vee \quad y' < \ldots < n \ \wedge\ x \in \mathrm{ipdom}_{\sqsupseteq}(n) \ \text{ for some } n$$

where

$$< \quad := \quad \{ (n, m) \mid n <^{C,k}_{\mathrm{TIME}} m,\ m \notin M \}$$

is obtained from $<^{C}_{\mathrm{TIME}}$ by deleting all edges into M.

The resulting Algorithm 25 differs from Algorithm 11 only by addition of the highlighted lines.

```
Input   : A CFG $G = (N, E)$
Input   : Any numbering $^\# : N \to \mathbb{N}$
Input   : Immediate post dominators $\text{ipdom}_{\sqsupseteq} = \text{ipdom}_{\sqsupseteq^C_{\text{TIME[FIRST]}}}$
Input   : The transitive reduction $<^C_{\text{TIME}}$ of transitive timing sensitive
          postdominance under timing cost model $C$
Input   : The set $\mathbb{M}$ of $<^C_{\text{TIME}}$-cycles $M$
Input   : The set $N_M$ of corresponding "border" nodes in $<^C_{\text{TIME}}$, and
          $N_{\mathbb{M}} = \bigcup_{M \in \mathbb{M}} N_M$
Data: A priority queue Q ordered by the numbering $^\#$
Output: $\text{PDF}_{\sqsupseteq^C_{\text{TIME[FIRST]}}}$ represented as a map $\text{DF} : N \to N \hookrightarrow \text{Bool}$
$\text{Q} \leftarrow \emptyset$
for $M \in \mathbb{M}, |M| > 1$ do
    for $y \in N_M, x \in M \setminus \text{ipdom}_{\sqsupseteq}(y),\ \exists y'.\ y \to_G y' \wedge x \sqsupseteq y'$ do
        $\text{DF}[x][y] \leftarrow$ false
    end
end
for $x \in N,\ y \to_G x,\ \neg\, x \in \text{ipdom}_{\sqsupseteq}(y)$ do
    $\text{DF}[x][y] \leftarrow$ true
    $\text{Q} \leftarrow \text{Q} \cup \{x\}$
end
while $\text{Q} \neq \emptyset$ do
    $z \leftarrow \text{remove(Q) s.t. } z^\# = \max_{z \in Q} z^\#$
    for $x \in \text{ipdom}_{\sqsupseteq}(z),\ (y, \textbf{true}) \in \text{DF}[z],\ \neg\, x \in \text{ipdom}_{\sqsupseteq}(y)$ do
        $\text{DF}_{x,y} \leftarrow \text{DF}[x][y]$
        $\text{DF}'_{x,y} \leftarrow \text{DF}_{x,y} \vee (z \notin N_{\mathbb{M}})$
        if $\text{DF}'_{x,y} \neq \text{DF}_{x,y}$ then
            $\text{DF}[x][y] \leftarrow \text{DF}'_{x,y}$
            $\text{Q} \leftarrow \text{Q} \cup \{x\}$
        end
    end
end
return DF
```

Algorithm 25: Computation of $\text{PDF}_{\sqsupseteq^C_{\text{TIME[FIRST]}}}$

```
Input  : A CFG $G$
Input  : A timing cost model $C$ for $G$
Data: A $\mathbb{N}$ labeled pseudo-forest $<$, represented as a map
      $\mathsf{IDOM} : N \hookrightarrow N \times \mathbb{N}$ s.t. $\mathsf{IDOM}[n] = (m, k)$ iff $n <^k m$
Output: A transitive reduction $>^C_{\mathrm{TIME}}$ of $\sqsupseteq^C_{\mathrm{TIME}}$
begin
    for $x \in N,\ \{z \mid x \to_G z\} = \{z\}$ do
        $\mathsf{IDOM}[x] \leftarrow \left(z, \boxed{C(x,z)}\right)$
    end
    $\mathrm{TIME_{up}}$
    return $\mathsf{IDOM}$
end
Procedure $\mathrm{TIME_{up}}$
    $\mathsf{workset} \leftarrow \mathrm{COND}_G$
    while $\mathsf{workset} \neq \emptyset$ do
        $x \leftarrow remove(\mathsf{workset})$
        $(z,k) \leftarrow \mathsf{lca}_<\left(\left\{\left(y, \boxed{C(x,y)}\right) \mid x \to_G y\right\}\right)$
        assert $(z,k) \neq \mathsf{IDOM}[x] \Rightarrow (z,k) \neq \bot$
        assert $(z,k) \neq \mathsf{IDOM}[x] \Rightarrow \mathsf{IDOM}[x] = \bot$
        if $(z,k) \neq \mathsf{IDOM}[x]$ then
            $\mathsf{workset} \leftarrow \mathsf{workset} \cup \{n \in \mathrm{COND} \mid n \neq x, \exists n \to_G y.\ y <^* x\}$
            $\mathsf{IDOM}[x] \leftarrow (z,k)$
        end
    end
end
```

Algorithm 26: An efficient algorithm for the computation of $<^C_{\mathrm{TIME}}$. Here, $y <^* x$ is taken to mean: $x = y \ \vee\ y <^{k_1} \ldots <^{k_c} x$

Bibliography

Ahenny (adj.) — The way people stand when examining other people's bookshelves.

(Douglas Adams — The Deeper Meaning of Liff)

[Aga00] Johan Agat. "Transforming out Timing Leaks". In: *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '00. Boston, MA, USA: ACM, 2000, pp. 40–53. ISBN: 1-58113-125-9. DOI: `10.1145/325694.325702`.

[Alm+16] Jose Bacelar Almeida et al. "Verifying Constant-Time Implementations". In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 53–70. ISBN: 978-1-931971-32-4.

[Amt08] Torben Amtoft. "Slicing for modern program structures: a theory for eliminating irrelevant loops". In: *Information Processing Letters* 106.2 (2008), pp. 45–51. ISSN: 0020-0190. DOI: `https://doi.org/10.1016/j.ipl.2007.10.002`.

[AS85] Bowen Alpern and Fred B. Schneider. "Defining liveness". In: *Information Processing Letters* 21.4 (1985), pp. 181–185. ISSN: 0020-0190. DOI: `https://doi.org/10.1016/0020-0190(85)90056-0`.

[Ask+08] Aslan Askarov et al. "Termination-Insensitive Noninterference Leaks More Than Just a Bit". In: *Computer Security - ESORICS 2008*. Ed. by Sushil Jajodia and Javier Lopez. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 333–348. ISBN: 978-3-540-88313-5.

[Bar+10] Richard W. Barraclough et al. "A trajectory-based strict semantics for program slicing". In: *Theoretical Computer Science* 411.11 (2010), pp. 1372–1386. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/j.tcs.2009.10.025`.

[BB05] David Brumley and Dan Boneh. "Remote timing attacks are practical". In: *Computer Networks* 48.5 (2005). Web Security, pp. 701–716. ISSN: 1389-1286. DOI: `https://doi.org/10.1016/j.comnet.2005.01.010`.

[Ber05] Daniel J. Bernstein. *Cache-timing attacks on AES*. Tech. rep. 2005.

[Bis+18a] Simon Bischof et al. "Illi Isabellistes Se Custodes Egregios Praestabant". In: *Principled Software Development: Essays Dedicated to Arnd Poetzsch-Heffter on the Occasion of his 60th Birthday*. Ed. by Peter Müller and Ina Schaefer. Springer International Publishing, 2018, pp. 267–282. DOI: `10.1007/978-3-319-98047-8_17`.

[Bis+18b] Simon Bischof et al. "Low-Deterministic Security For Low-Nondeterministic Programs". In: *Journal of Computer Security* 26 (2018), pp. 335–366. DOI: `10.3233/JCS-17984`.

[Bis19] Simon Bischof. *Isabelle Theories about NTxCD*. 2019. URL: `https://pp.ipd.kit.edu/~bischof/ntxcd.html` (visited on 12/13/2010).

[BJ66] Corrado Böhm and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". In: *Commun. ACM* 9.5 (May 1966), pp. 366–371. ISSN: 0001-0782. DOI: `10.1145/355592.365646`.

[BM06] Joseph Bonneau and Ilya Mironov. "Cache-Collision Timing Attacks Against AES". In: *Cryptographic Hardware and Embedded Systems - CHES 2006*. Ed. by Louis Goubin and Mitsuru Matsui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 201–215. ISBN: 978-3-540-46561-4.

[BP10] Joan Boyar and René Peralta. "A New Combinational Logic Minimization Technique with Applications to Cryptology". In: *Experimental Algorithms*. Ed. by Paola Festa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 178–189. ISBN: 978-3-642-13193-6.

[BP96] Gianfranco Bilardi and Keshav Pingali. “A Framework for Generalized Control Dependence”. In: *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*. PLDI ’96. Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 291–300. ISBN: 0-89791-795-2. DOI: `10.1145/231379.231435`.

[Bre+16] Joachim Breitner et al. “On Improvements Of Low-Deterministic Security”. In: *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Frank Piessens and Luca Viganò. Vol. 9635. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2016, pp. 68–88. DOI: `10.1007/978-3-662-49635-0_4`.

[Bro+19] R. Brotzman et al. “CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. May 2019, pp. 505–521. DOI: `10.1109/SP.2019.00022`.

[BRW06] Gilles Barthe, Tamara Rezk, and Martijn Warnier. “Preventing Timing Leaks Through Transactional Branching Instructions”. In: *Electronic Notes in Theoretical Computer Science* 153.2 (2006). Proceedings of the Third Workshop on Quantitative Aspects of Programming Languages (QAPL 2005), pp. 33–55. ISSN: 1571-0661. DOI: `https://doi.org/10.1016/j.entcs.2005.10.031`.

[BS07] Julian Bradfield and Colin Stirling. “12 Modal mu-calculi”. In: *Handbook of Modal Logic*. Ed. by Patrick Blackburn, Johan Van Benthem, and Frank Wolter. Vol. 3. Studies in Logic and Practical Reasoning. Elsevier, 2007, pp. 721–756. DOI: `https://doi.org/10.1016/S1570-2464(07)80015-2`.

[CFT03] Larry Carter, Jeanne Ferrante, and Clark Thomborson. “Folklore Confirmed: Reducible Flow Graphs Are Exponentially Larger”. In: *Proceedings of the 30th ACM*

SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL ’03. New Orleans, Louisiana, USA: ACM, 2003, pp. 106–114. ISBN: 1-58113-628-5. DOI: `10.1145/604131.604141`.

[CH02] Koen Claessen and John Hughes. “Testing Monadic Code with QuickCheck”. In: *SIGPLAN Not.* 37.12 (Dec. 2002), pp. 47–59. ISSN: 0362-1340. DOI: `10.1145/636517.636527`.

[Cha+19] Sudipta Chattopadhyay et al. “Quantifying the Information Leakage in Cache Attacks via Symbolic Execution”. In: *ACM Trans. Embed. Comput. Syst.* 18.1 (Jan. 2019). ISSN: 1539-9087. DOI: `10.1145/3288758`.

[CHK01] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. “A simple, fast dominance algorithm”. In: *Software Practice & Experience* (2001).

[Cor20] Oracle Corporation. *Code Tools: jmh*. 2020. URL: `https://github.com/AlDanial/cloc` (visited on 02/11/2020).

[CR06] Feng Chen and Grigore Roşu. “Parametric and Termination-Sensitive Control Dependence”. In: *Static Analysis*. Ed. by Kwangkeun Yi. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 387–404. ISBN: 978-3-540-37758-0.

[CS10] Michael R Clarkson and Fred B Schneider. “Hyperproperties”. In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210.

[Cyt+91] Ron Cytron et al. “Efficiently Computing Static Single Assignment Form and the Control Dependence Graph”. In: *ACM Trans. Program. Lang. Syst.* 13.4 (Oct. 1991), pp. 451–490. ISSN: 0164-0925. DOI: `10.1145/115372.115320`.

[Dan+05] Sebastian Danicic et al. “Static program slicing algorithms are minimal for free liberal program schemas”. In: *The Computer Journal* 48.6 (2005), pp. 737–748.

[Dan+11] Sebastian Danicic et al. “A unifying theory of control dependence and its application to arbitrary program structures”. In: *Theoretical Computer Science* 412.49 (2011), pp. 6809–6842. ISSN: 0304-3975. DOI: `https://doi.org/10.1016/j.tcs.2011.08.033`.

[Dan18] Al Danial. *Cloc*. 2006 - 2018. URL: `https://github.com/AlDanial/cloc` (visited on 02/07/2020).

[Doy+15] Goran Doychev et al. “CacheAudit: A Tool for the Static Analysis of Cache Side Channels”. In: *ACM Trans. Inf. Syst. Secur.* 18.1 (June 2015). ISSN: 1094-9224. DOI: `10.1145/2756550`.

[EM17] Martin Erwig and Ivan Lazar Miljenovic. *fgl: Martin Erwig's Functional Graph Library*. `https://hackage.haskell.org/package/fgl`. 2001–2017.

[Erw01] Martin Erwig. “Inductive graphs and functional graph algorithms”. In: *Journal of Functional Programming* 11.5 (2001), pp. 467–492. DOI: `10.1017/S0956796801004075`.

[Fou19] The Apache Software Foundation. *Apache FtpServer*. 2019. URL: `https://mina.apache.org/ftpserver-project/download_1.1.1.html` (visited on 02/07/2020).

[FOW87] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. “The Program Dependence Graph and Its Use in Optimization”. In: *ACM Trans. Program. Lang. Syst.* 9.3 (July 1987), pp. 319–349. ISSN: 0164-0925. DOI: `10.1145/24039.24041`.

[Fri02] Carsten Fritz. “Some Fixed Point Basics”. In: *Automata Logics, and Infinite Games: A Guide to Current Research*. Ed. by Erich Grädel, Wolfgang Thomas, and Thomas Wilke. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 359–364. ISBN: 978-3-540-36387-3. DOI: `10.1007/3-540-36387-4_20`.

[Gaw+11] Thomas Martin Gawlitza et al. “Join-Lock-Sensitive Forward Reachability Analysis for Concurrent Programs with Dynamic Process Creation”. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Ranjit Jhala and David Schmidt. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 199–213. ISBN: 978-3-642-18275-4.

[Gif12] Dennis Giffhorn. “Slicing of Concurrent Programs and its Application to Information Flow Control”. PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, 2012.

[GM03] Roberto Giacobazzi and Isabella Mastroeni. “Non-Standard Semantics for Program Slicing”. In: *Higher-Order and Symbolic Computation* 16.4 (Dec. 2003), pp. 297–339. ISSN: 1573-0557. DOI: `10.1023/A:1025872819613`.

[Gra16] Jürgen Graf. “Information Flow Control with System Dependence Graphs – Improving Modularity, Scalability and Precision for Object Oriented Languages”. PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, Nov. 2016. DOI: `10.5445/IR/1000068211`.

[GS15] Dennis Giffhorn and Gregor Snelting. “A new algorithm for low-deterministic security”. In: *International Journal of Information Security* 14.3 (2015), pp. 263–287. DOI: `10.1007/s10207-014-0257-6`.

[Ham09] Christian Hammer. “Information Flow Control for Java - A Comprehensive Approach based on Path Conditions in Dependence Graphs”. ISBN 978-3-86644-398-3. PhD thesis. Universität Karlsruhe (TH), Fak. f. Informatik, July 2009.

[Hec20] Martin Hecker. *Properties for: Timing Sensitive Dependency Analysis and its Application to Software Security*. 2020. URL: `https://pp.ipd.kit.edu/~hecker/dissertation/` (visited on 03/03/2010).

[HS09] Christian Hammer and Gregor Snelting. “Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs”. In: *International Journal of Information Security* 8.6 (Dec. 2009), pp. 399–422. ISSN: 1615-5270. DOI: 10.1007/s10207-009-0086-1.

[HU73] Matthew S. Hecht and Jeffrey D. Ullman. “Analysis of a Simple Algorithm for Global Data Flow Problems”. In: *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL ’73. Boston, Massachusetts: ACM, 1973, pp. 207–217. DOI: 10.1145/512927.512946.

[Jus+11] Seth Just et al. “Information Flow Analysis for Javascript”. In: *Proceedings of the 1st ACM SIGPLAN International Workshop on Programming Language and Systems Technologies for Internet Clients*. PLASTIC ’11. Portland, Oregon, USA: ACM, 2011, pp. 9–18. ISBN: 978-1-4503-1171-7. DOI: 10.1145/2093328.2093331.

[Koc+19] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *40th IEEE Symposium on Security and Privacy (S&P’19)*. 2019.

[KRB09] B. Köpf, A. Rybalchenko, and M. Backes. “Automatic Discovery and Quantification of Information Leaks”. In: *2009 30th IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2009, pp. 141–153. DOI: 10.1109/SP.2009.18.

[Kri98] Jens Krinke. “Static Slicing of Threaded Programs”. In: *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. PASTE ’98. Montreal, Quebec, Canada: ACM, 1998, pp. 35–42. ISBN: 1-58113-055-4. DOI: 10.1145/277631.277638.

[KTG12] Ralf Küsters, Tomasz Truderung, and Jürgen Graf. “A Framework for the Cryptographic Verification of Java-like Programs”. In: *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*. IEEE Computer Society, June 2012.

[Küs+14] Ralf Küsters et al. “Extending and Applying a Framework for the Cryptographic Verification of Java Programs”. In: *Proc. POST 2014*. LNCS 8424. Springer, 2014, pp. 220–239.

[Lip+18] Moritz Lipp et al. “Meltdown: Reading Kernel Memory from User Space”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[LKL18] Jean-Christophe Léchenet, Nikolai Kosmatov, and Pascale Le Gall. “Fast Computation of Arbitrary Control Dependencies”. In: *Fundamental Approaches to Software Engineering*. Ed. by Alessandra Russo and Andy Schürr. Cham: Springer International Publishing, 2018, pp. 207–224. ISBN: 978-3-319-89363-1.

[Lor+14] Steffen Lortz et al. “Cassandra: Towards a Certifying App Store for Android”. In: *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones &#38; Mobile Devices*. SPSM ’14. Scottsdale, Arizona, USA: ACM, 2014, pp. 93–104. ISBN: 978-1-4503-3155-5. DOI: `10.1145/2666620.2666631`.

[LT79] Thomas Lengauer and Robert Endre Tarjan. “A Fast Algorithm for Finding Dominators in a Flowgraph”. In: *ACM Trans. Program. Lang. Syst.* 1.1 (Jan. 1979), pp. 121–141. ISSN: 0164-0925. DOI: `10.1145/357062.357071`.

[Mil17] Ivan Lazar Miljenovic. *fgl-arbitrary: QuickCheck support for fgl*. `https://hackage.haskell.org/package/fgl-arbitrary`. 2015–2017.

[Mol+06] David Molnar et al. “The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks”. In: *Information Security and Cryptology - ICISC 2005*. Ed. by Dong Ho Won and Seungjoo Kim. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 156–168. ISBN: 978-3-540-33355-5.

[Mou+15] Leonardo de Moura et al. "The Lean Theorem Prover (System Description)". In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 378–388. ISBN: 978-3-319-21401-6.

[MS15] Heiko Mantel and Artem Starostin. "Transforming Out Timing Leaks, More or Less". In: *Computer Security – ESORICS 2015*. Ed. by Günther Pernul, Peter Y A Ryan, and Edgar Weippl. Cham: Springer International Publishing, 2015, pp. 447–467. ISBN: 978-3-319-24174-6.

[NP19] Barak Naveh and Stephane Popinet. *JGraphT: A Java Library of Graph Theory Data Structures and Algorithms*. `https://jgrapht.org/`. 2003–2019.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Berlin, Heidelberg: Springer-Verlag, 2002. ISBN: 3-540-43376-7.

[PC90] A. Podgurski and L. A. Clarke. "A formal model of program dependences and its implications for software testing, debugging, and maintenance". In: *IEEE Transactions on Software Engineering* 16.9 (Sept. 1990), pp. 965–979. ISSN: 0098-5589. DOI: `10.1109/32.58784`.

[PP18] Amanda Plunkett and Junyong Park. "Two-sample test for sparse high-dimensional multinomial distributions". In: *TEST* (July 2018). ISSN: 1863-8260. DOI: `10.1007/s11749-018-0600-8`.

[Ran+07] Venkatesh Prasad Ranganath et al. "A New Foundation for Control Dependence and Slicing for Modern Program Structures". In: *ACM Trans. Program. Lang. Syst.* 29.5 (Aug. 2007). ISSN: 0164-0925. DOI: `10.1145/1275497.1275502`.

[SG95] Vugranam C. Sreedhar and Guang R. Gao. "A Linear Time Algorithm for Placing Φ-nodes". In: *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '95. San Francisco, California, USA: ACM, 1995, pp. 62–73. ISBN: 0-89791-692-1. DOI: `10.1145/199448.199464`.

[Sne+14] Gregor Snelting et al. "Checking Probabilistic Noninterference Using JOANA". In: *it - Information Technology* 56 (Nov. 2014), pp. 280–287. DOI: `10.1515/itit-2014-1051`.

[Tar72] Robert Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM Journal on Computing* 1.2 (1972), pp. 146–160. DOI: `10.1137/0201010`.

[Tea17] The Coq Development Team. *The Coq Proof Assistant, version 8.7.0*. Version 8.7.0. Oct. 2017. DOI: `10.5281/zenodo.1028037`.

[Was10] Daniel Wasserrab. "From Formal Semantics to Verified Slicing - A Modular Framework with Applications in Language Based Security". PhD thesis. Karlsruher Institut für Technologie, Fakultät für Informatik, Oct. 2010.

[Wei81] Mark Weiser. "Program Slicing". In: *Proceedings of the 5th International Conference on Software Engineering*. ICSE '81. San Diego, California, USA: IEEE Press, 1981, pp. 439–449. ISBN: 0-89791-146-6.

[Wel10] Stefan Wellek. *Testing Statistical Hypotheses of Equivalence and Noninferiority*. 2nd ed. Boca Raton, FL, USA: CRC Press, 2010. ISBN: 978-1-4398-0819-1.

[Wol95] Michael Joseph Wolfe. *High Performance Compilers for Parallel Computing*. Ed. by Carter Shanklin and Leda Ortega. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0805327304.

[Wu+18] Meng Wu et al. “Eliminating Timing Side-channel Leaks Using Program Repair”. In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: ACM, 2018, pp. 15–26. ISBN: 978-1-4503-5699-2. DOI: `10.1145/3213846.3213851`.

[XZ07] Bin Xin and Xiangyu Zhang. “Efficient Online Detection of Dynamic Control Dependence”. In: *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ISSTA ’07. London, United Kingdom: ACM, 2007, pp. 185–195. ISBN: 978-1-59593-734-6. DOI: `10.1145/1273463.1273489`.

Beppu (n.) — The triumphant slamming shut of a book after reading the final page.

(Douglas Adams — The Deeper Meaning of Liff)