A GENERAL FRAMEWORK FOR THE FLOW ANALYSIS OF CONCURRENT PROGRAMS

by

Patrick Lam

School of Computer Science

McGill University, Montreal

August 2000

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES AND RESEARCH IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE

Abstract

Standard techniques for analysing sequential programs are severely constrained when applied to a concurrent program because they cannot take full advantage of the concurrent structure of the program. In this work, we overcome this limitation using a novel approach which "lifts" a sequential dataflow analysis to a concurrent analysis. First, we introduce concurrency primitives which abstract away from the details of how concurrency features are implemented in real programming languages. Using these primitives, we describe how sequential analyses can be made applicable to concurrent programs. Under some circumstances, there is no penalty for concurrency: our method produces results which are as precise as the sequential analysis. Our lifting is straightforward, and we illustrate it on some standard analyses – available expressions, live variables and generalized constant propagation. Finally, we describe how concurrency features of real languages can be expressed using our abstract concurrency primitives, and present analyses for finding our concurrency primitives in real programs.

Résumé

Les méthodes généralement utilisées pour analyser les programmes sequentiels sont très limitées lorsqu'appliquées aux programmes construits avec des processus indépendents: elles ne peuvent se servir pleinement de la structure partiellement ordonnée des élements du programme. Nous proposons ici une approche originale qui dépasse ces limites en permettant d'appliquer des méthodes d'analyses standards aux programmes parallèles. Nous commençons par introduire des primitives qui nous permettent de faire abstraction des différentes implantations de synchronizations et de démarrage de processus indépendants. Grâces à ces primitives nous décrivons comment des analyses séquentielles peuvent être appliquées aux programmes simultanés. Nous démontrons que, dans certains cas, les résultats de ces analyses sont aussi précis que ceux d'analyses classiques. Notre méthode de généralisation d'analyse est simple. Nous en donnons quelques exemples: ≪available expressions≫, ≪live variables≫ et ≪generalized constant propagation≫. Finalement, nous décrivons comment les primitives de quelques langages de programmation correspondent aux primitives que nous avons présentés, et proposons des algorithmes pour décerner ces dernières.

Acknowledgements

My advisor, Prakash Panangaden, deserves many thanks. His support and guidance made this thesis possible. His dedication to students – not only his own graduate students, but also to undergraduate students in his classes – must be gratefully acknowledged. McGill University is fortunate to have him on faculty. Prakash has given me a glimpse at how math is really done.

Laurie Hendren taught me about compilers and flow analysis and other things practical. Staying well grounded in practice definitely improved my work.

At times, it was less than obvious that I would finish my thesis on time. The "July 17th Club": Theodoro Koulis, Marcus Hum and Raja Vallée-Rai, reminded me that I was not the only one in this situation. Raja, we must ensure that Soot achieves world domination.

The denizens of the Society of Undergraduate Mathematics Students (SUMS) lounge were very helpful during this work; many productive hours were spent in SUMS[1]. In particular, I must thank Peter Green for helping me rework the introduction and conclusion until they became suitable. I also wish to thank Marcus Hum for helping me make the correct aesthetic choices for the layout of this thesis.

I will always be indebted to my parents. I can only hope that I will be one day as successful as they have been at the difficult task of raising a child.

Finally, I am lucky to know Marie-Pascale. Her love, support and companionship have been invaluable. I will always treasure the moments I have spent with her.

[1] Playing `xblast`.

This work was supported by the Natural Sciences and Engineering Research Council and the Fonds pour la Formation de Chercheurs et l'Aide à la Recherche.

Contents

List of Figures

Chapter 1

Introduction

Concurrent programs exist, and in increasing numbers. Compiler techniques have traditionally focussed on improving sequential programs. In this work, we provide a general method for making analyses of sequential programs applicable to concurrent programs. Our method is general in the sense that it is applicable to a large number of sequential analyses; furthermore, our examples will show that our method is quite simple to apply. Using our techniques, compilers will also be able to improve concurrent programs.

Two recent trends have accelerated the development of concurrent programs. Symmetric multi-processor (SMP) computers are becoming increasingly available; writers of concurrent programs hope that parallelism will make their programs execute more quickly. At the same time, concurrency primitives are appearing in mainstream programming languages such as Java; they have long been present in Ada and CML. Programmers may use concurrency hoping for improved performance from parallelism, or alternately because it is the best way to solve a problem at hand. The latter phenomenon occurs when programming window systems, for example; each window corresponds readily to a separate thread. In any case, compiler writers must account for the effects of concurrency to correctly translate, and optimize, these programs.

In this work, we will deal with concurrent programs. There is a large body of work examining how to extract parallelism from ordinary sequential programs, in order to speed them up. Our study is orthogonal to this body of work. In our case, we have a program explicitly containing concurrency primitives, and we wish to analyse it.

The key difference between sequential programs and concurrent programs is that sequential programs have a single thread of execution and a total ordering between statements, while concurrent programs have many threads of execution and a partial ordering between statements. On appropriate hardware, threads may actually execute in parallel; in any case, execution of the various threads is certainly interleaved.

Compilers usually analyse programs in order to deduce aspects of their run-time behaviour. This allows a compiler to produce target code which run more quickly, or take up less space, than the naive translation. This process is known as optimization. There is a vast body of work concerning optimizations for sequential programs; there is relatively little work on optimizing concurrent programs. This thesis presents a general method for lifting sequential analyses to concurrent programs.

Many program analyses are flow-sensitive; that is, they use the execution ordering of instructions in programs to deduce facts about these programs. In order to apply flow-sensitive analyses to multithreaded programs, we must use the partial ordering between instructions. At our disposal, we have information about inter-thread communication, in terms of primitives scattered through the program. These primitives – fork, join, and synchronization – affect the order in which instructions are executed.

To carry out flow-sensitive analyses, we must estimate the flow of control. For sequential programs, the major source of imprecision in this estimate is due to deterministic choice. In general, we do not estimate which choices will be taken at runtime; instead, we conservatively assume that each branch is possible. In a concurrent program, the interleaving of instructions introduces another sort of imprecision into the estimate of the flow of control.

1.1 Thesis Contributions

Traditional compiler frameworks are generally ineffective for optimizing concurrent programs because they must make much too conservative assumptions about the effects of concurrency. The main contribution of this thesis is to provide a compiler framework which overcomes these limitations and allows the optimization of concurrent programs. This is achieved by proposing a general lifting for sequential dataflow analyses: given a sequential analysis, we describe how an equivalent, "lifted", analysis for concurrent programs may be carried out. For many dataflow analyses, this lifting is the best possible one.

Our lifting is described using a set of abstract concurrency primitives, so that many languages can be analysed. We present analyses to detect these primitives for several programming languages, like Java and Ada.

To provide the new optimization framework, the thesis makes the following original contributions:

- We introduce concurrency primitives which are more expressive than the simple fork and join primitives considered in earlier work. In particular, they can handle mutual exclusion and synchronization. Conservative assumptions were used in previous work for these primitives; our lifting deals with them more effectively in a sense that we will make precise.

- We provide techniques for the detection of statements which can happen in parallel, for Java programs with procedures, and inexact information about synchronization. Previous work only considered the case of Java programs without procedures and exact synchronization information.

- Using this infrastructure, the main contribution of this work is a general lifting for sequential dataflow analyses. Given a sequential dataflow analysis, we will provide the concurrent analogue. Under some conditions, we provide the best possible analogue. This lifting behaves well in the presence of our expressive concurrency primitives.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. We first present classical material about temporal ordering, program traces and standard sequential flow analysis in Chapter 2. Next, we discuss traces of concurrent programs in Chapter 3; we describe how causal relations between statements affects the set of traces, and we provide a description of how some simple concurrency primitives map into our causal relations. We briefly treat the effects of approximation, and we also provide an example showing why traditional sequential techniques cannot be applied. The central part of our work is in Chapter 4; we describe the lifting of sequential analyses to the concurrent setting and prove that it has several desirable properties. Some examples of liftings are described in Chapter 5; in particular, we lift the available expressions, live variables and generalized constant propagation analyses. We return to the topic of traces in Chapter 6, showing how our analyses are applicable to real Java and Ada programs. Finally, we discuss related work in Chapter 7 and conclude in Chapter 8.

Chapter 2

Background

The background material splits into three parts. The first part is, by now, classical material from distributed systems and derives from the fundamental paper, "Time, Clocks and the Ordering of Events in a Distributed System" by Leslie Lamport [Lam77]. Next, we introduce a notation for discussing program traces. Finally, we present material from sequential data-flow analysis; it is included for completeness, and also to adapt the terminology to our concurrent setting.

2.1 Causal and temporal ordering

In any computational (or physical) system the fundamental unit of activity is the *event*. In a simple system all events are totally ordered and the ordering is *temporal precedence*. This order is unambiguous because the effect of an event is instantaneous. In a distributed system we have the notion of *location* as a site of activity. The effect of an event at one location does not propagate instantly to other locations. Thus the notion of temporal precedence between events loses its absolute significance; what survives as an absolute notion is *causal precedence*.

In order to set up a general theory of concurrent systems we have to work with the causality relation. An essential part of concurrency is that one deals with autonomous computing agents or, what amounts to the same thing, different loci of activity. It is not just that there are multiple threads of control, there is a delay in the interaction between threads.

This clean picture was enunciated by Lamport using a close analogy with the notion of causal structure in relativity [Wal84]. In our case, however, there is a further complication. Threads – unlike processes communicating via messages – may share state. Thus the separation into local and global is less clear cut. Some effects are felt instantly and some are felt through the mediation of synchronization mechanisms and are delayed.

We define a causal relation between events that reflects the effects of interaction through synchronization mechanisms and ignores the effects of shared state. This still determines a sensible partial ordering on events but the name "causal order" is now less apt; the phrase "happens before" popularized by Lamport is more appropriate. Nevertheless this partial order that we define is still the determinant of temporal precedence. The possible temporal orders are captured as the set of all linearizations of the partial order. The effect of immediate interaction through the shared state will manifest itself through the equations for computing flow information.

2.1.1 The happens-before relation

Let us assume that events can be unambiguously associated with threads. An *execution* of a multithreaded program consists of a vector of sequences of events indexed by the threads. Thus, for each thread we have a sequence of events; this reflects the idea that at a single locus of activity the events are totally ordered. If there were no synchronization mechanism and no interaction between threads this would be the complete temporal structure. Suppose that we have a synchronization mechanism that enforces a temporal precedence between an event x in one thread and an event y in another thread. This could take the form of a wait-notify pair in Java, a rendezvous in Ada, or a send-receive pair in any message-passing formalism. Whatever the mechanism, we posit a primitive binary relation denoted $\twoheadrightarrow$. This relation is called "happens immediately before".

Restricted to a thread, $\twoheadrightarrow$ is just the immediate temporal precedence order. It is irreflexive.

In a language like CML or Occam where communication is synchronous one can have $x \twoheadrightarrow y$ and $y \twoheadrightarrow x$ but in languages with asynchronous communication this will never happen.

Definition 2.1.1 *We define the* ***happens-before*** *relation, written $x \rightarrow y$, as the reflexive transitive closure of $\twoheadrightarrow$.*

Because we could have a situation where $x \twoheadrightarrow y$ and $y \twoheadrightarrow x$ it will not be true, in general, that $\rightarrow$ is a partial order. We say that a cycle in $\rightarrow$ is *trivial* if it is also a cycle of $\twoheadrightarrow$ and involves just two events. We say that an execution is *causal* if there are no nontrivial cycles in $\rightarrow$.

We assume henceforth (until chapter 6) that no cycles occur.

We point out that neither $x \twoheadrightarrow y$ nor $x \rightarrow y$ guarantee that if x executes, then y executes. Nor does it guarantee that if y occurs than x occurred before. If both x and y occur, then x occurs before y.

If $x \twoheadrightarrow y$ then we specifically do not require any temporal relation between the successors of x and anything in y's thread.

Definition 2.1.2 *A* trace *of a program execution is a sequence of all the events from an execution of the program, ordered in such a way as to extend $\rightarrow$.*

Recall that in any trace of the program execution with both x and y, $x \rightarrow y$ means that x occurs before y in any trace.

Definition 2.1.3 *If $x \not\rightarrow y$ and $y \not\rightarrow x$, then we say that x and y are* independent *($x \parallel y$).*

This means that x does not necessarily precede y and vice-versa. If there were no shared state one could also say that $x \parallel y$ means that x and y do not influence or affect each other, but this is definitely not true in our case.

A trace of a sequential program is the semantic entity that is used for all issues of correctness of a flow analysis. It says "what actually happens" in a program execution. In the case of concurrent programs this is still true, but a trace, as we have defined it, contains too much incidental temporal information. It turns out that it will be much more convenient to have as the basic semantic entity the poset defined by $\rightarrow$.

In a sequential program, resolving the conditional branches gives a trace of the execution. In a multithreaded program, there are still conditional branches, but the trace also depends on the interleaving of the executions of various threads, at the scheduler's whim. We resolve the deterministic choice by selecting a branch from each conditional. We are left with a set of events (program points) in various threads organized into a vector of sequences as defined above. The $\rightarrow$ relation partially orders the events.

Remark about sequential merges We will emphasize that $\twoheadrightarrow$ specifically does not apply at sequential merges of control. This is because $\twoheadrightarrow$ applies between events in the same trace; it does not relate events in different traces.

Definition 2.1.4 *A* scenario[1] *is the poset of events in a program execution, related by* $\rightarrow$*.*

A diagrammatic representation of a scenario showing time proceeding horizontally, thread executions as horizontal lines stacked vertically, and instances of $\twoheadrightarrow$ as arrows across threads is called a *Lamport diagram*; see, for example, Figure 2.1. In a scenario, no program point is executed more than once; there are no "loops", as different executions of the same program point are represented as different events[2].

[1] This term was used by Brock and Dennis in an early formalization of concurrency semantics.

[2] If we viewed different executions of the same program point as the same event we would have a multiset of events with the causal structure; in short we would have a pomset.

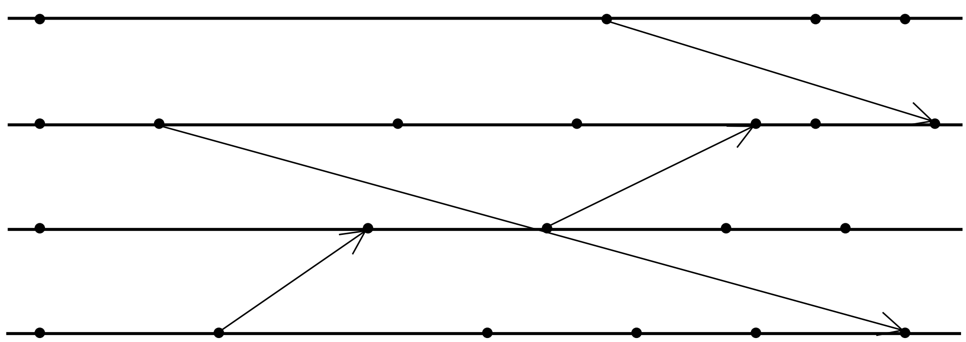


Figure 2.1: A typical Lamport diagram

2.2 Execution sequences

In the previous section, we have discussed the trace of a program's execution. We will now introduce a notation for these traces. This notation is useful for reasoning about the flow of control in both sequential and concurrent programs.

Definition 2.2.1 *A* configuration *is a pair* (s, σ)*, where* s *is a program point, and* σ *is a store.*

Definition 2.2.2 *A* control trace *is a sequence*

$$s_0, s_1, \ldots, s_n, \ldots$$

where $\forall i.$ *there is an edge from* s_i *to* s_{i+1} *in the control-flow graph. We denote the set of all control traces as* $\mathcal{P}$*.*

Definition 2.2.3 *An* actual execution trace e *is a sequence*

$$(s_0, \sigma_0), (s_1, \sigma_1), \ldots, (s_n, \sigma_n), \ldots$$

where $\forall i.$ $(s_i, \sigma_i) \rightarrow (s_{i+1}, \sigma_{i+1})$ *in the operational semantics. We denote by* $\mathcal{E}$ *the set of all actual execution traces. We write* $\Pi(e)$ *for the projection of* e *obtained by removing the stores. Similarly we write* $\Pi(\mathcal{E})$ *for the projection of every sequence in* $\mathcal{E}$*.*

For sequential programs, an actual execution trace is completely determined by the inputs; however, this is not true for concurrent programs. In a concurrent program, many actual execution traces may arise from one set of inputs (because many interleavings of statements from various threads are possible.)

It is very difficult to reason about the actual set of executions, as it is undecidable to determine whether or not a given statement appears in any actual execution. We will introduce an approximation to the actual set of executions.

Definition 2.2.4 *A* plausible set *of executions E is any set such that $\mathcal{P} \supseteq E \supseteq \Pi(\mathcal{E})$. We also define a* plausible graph G_E *corresponding to E by taking the edges in G which correspond to transitions $s_n \to s_{n+1}$ in E.*

The point of introducing plausible executions is that they represent an easy-to-define set of executions that are guaranteed to contain the projected actual executions. It is thus easier to reason about program behaviour with plausible executions.

The standard set of plausible executions used in dataflow analysis is the set of all control traces. If we can remove never-visited edges from G, we get a better set of plausible executions.

An execution will, in general, contain many instances of a given program point. We distinguish between them by using the index of the occurrence of a program point in the sequence.

Example 2.2.5 *Consider the following program.*

```
s:   i = 8;
t:   while (i < 10)
     {
u:      print("Counting");
v:      i++;
     }
```

Two plausible execution traces are (s, t, u, v) *and* (s, t, u, v, t, u, v)*. Of these two, only* (s, t, u, v, t, u, v) *is an actual execution trace.*

2.3 Sequential flow analysis

In this section we recapitulate the basic definitions and framework of sequential flow analysis as a preparation for our treatment of concurrent flow analysis, using the notation introduced in the previous section. The treatment here is not original, and is included for completeness. The classical works on flow analysis are by Kildall [Kil73], Kam and Ullman [KU77] and Cousot and Cousot [CC77]. Our treatment principally reviews the results of Kildall.

Typically, flow analysis is used to determine information about the run-time behaviour of a program. We must approximate the exact information because, for any nontrivial flow problem, it is undecidable to find exactly which dataflow facts would hold at any given program point.

The basic setup is that we have a graph representing the program (perhaps suitably abstracted) and we have a function that propagates information through this graph. We start with a crude approximation to the required flow information – one that will, in general, be valid only at the start node – and iterate to a fixed point by propagating the flow information through the graph. The approximation of the run-time information is represented in some abstraction domain, and it is with respect to this structure that we carry out the iteration to the fixed point. We require that the abstraction domain be a complete partial order with finite height. Usually, it turns out that the domain is a lattice, or even a powerset.

Definition 2.3.1 *A* dataflow analysis $\mathcal{F}$ *associates to each program point* $s \in G$ *two elements of a complete partial order* $\mathcal{D}$*. We can write:*

$$\begin{aligned} \mathcal{F}_{in} &: G \to \mathcal{D} \\ \mathcal{F}_{out} &: G \to \mathcal{D} \end{aligned}$$

We use the shorthand

$IN(s) := \mathcal{F}_{in}(s)$ *and* $OUT(s) := \mathcal{F}_{out}(s)$.

The IN set corresponds to run-time information about traces of the program just before some program point s is executed, while OUT corresponds to the state just after s executes.

Definition 2.3.2 *A flow function f is a G-indexed family of functions of type $\mathcal{D} \rightarrow \mathcal{D}$; that is,*

$$f : G \rightarrow (\mathcal{D} \rightarrow \mathcal{D})$$

For a forward analysis, given s and a purported value ℓ for $IN(s)$, $f_s(\ell)$ tells us $OUT(s)$; a backwards analysis similarly maps $OUT(s)$ to $IN(s)$.

We assume that flow functions are monotone: $x \sqsupseteq y \Rightarrow f(x) \sqsupseteq f(y)$.

Definition 2.3.3 *A flow function f is* multiplicative[3] *if $f(x \sqcap y) = f(x) \sqcap f(y)$; f is* additive *if $f(x \sqcup y) = f(x) \sqcup f(y)$.*

As long as the flow function is monotone, then a simple calculation gives us the following weaker properties:

$$f(x \sqcup y) \geq f(x) \sqcup f(y) \qquad f(x \sqcap y) \leq f(x) \sqcap f(y)$$

Definition 2.3.4 *(1) Given an abstracted actual sequence*

$$e = s_0 s_1 s_2 \cdots s_n \cdots$$

consider every occurrence of s in e; say $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$. We evaluate *$f$ on this sequence as follows:*

$$OUT_e(s) = \bigsqcup_{j}^{k} f_{s_{i_j}}(f_{s_{i_j}-1} \cdots f_{s_0}(\ell))$$

[3] In the classical references on flow analysis, this condition is known as distributivity. We use the correct terms here.

Let $t_1, \ldots, t_m$ be the statements preceding s in e. Then

$$IN_e(s) = \bigsqcup_{j=1}^{m} OUT_e(t_j)$$

(2) Given the set of actual execution sequences $\mathcal{S}$ we define the exact analysis modulo f *as*

$$OUT_{exact}(s) = \bigsqcup_{e \in \mathcal{S}} OUT_e(s)$$

(3) We write $\bigsqcup_{e \in \mathcal{S}} f(e)$ to denote the flow analysis assigning IN and OUT sets to each statement by taking the result of the exact analysis modulo f for that statement.

Example 2.3.5 *Returning to the program of example 2.2.5, if we assume that the two sequences (s_0, t_1, u_2, v_3) and $(s_0, t_1, u_2, v_3, t_4, u_5, v_6)$ are the only two actual execution sequences, then we would say that in the exact flow analysis modulo f, $IN(t) = f_s(\ell) \sqcup f_v(f_u(f_t(f_s(\ell))))$.*

An *approximate dataflow analysis* can be computed given a direction (we consider backward and forward flow analyses), a merge operator (e.g. join in a lattice), a flow function, a plausible graph and an element of the domain for the start node in the graph. The computation proceeds by evaluating the fixed point of the flow function over the graph. When the merge operator is join or meet, we denote this approximation by, respectively, $\mathrm{fix}_G(f, \sqcup)$ or $\mathrm{fix}_G(f, \sqcap)$.

Definition 2.3.6 *A* some-path *analysis is a dataflow analysis $\mathcal{F}$ such that for any $s \in G$ and any execution sequence e ending with s, $\mathcal{F}_{in}(s) \sqsupseteq OUT_e(s)$. An* all-paths *analysis is a dataflow analysis $\mathcal{F}$ such that $\forall s \in G$. $\forall$ execution sequences e ending with s, $\mathcal{F}_{in}(s) \sqsubseteq OUT_e(s)$.*

If the abstraction domain is a powerset, a some-paths analysis may omit no elements; an all-paths analysis may include no extraneous elements.

Observation A some-path analysis can be approximated using join ($\sqcup$) as the merge operator; an all-paths analysis can be approximated using meet ($\sqcap$) as the merge operator.

We now need to make precise what we mean by "evaluating the fixed point of the flow function over the graph".

First, label every edge in the control-flow graph: $h_1, h_2, \ldots, h_k$. The set of edges is called H. Consider the H-indexed vector of elements of the cpo; we call this $\vec{v}$; e.g. $\vec{v}[h_q]$ is an element ℓ of the cpo.

Definition 2.3.7 *We define the* graph flow function $\varphi : (H \to D) \to (H \to D)$ *for some-path analyses:*

$$\varphi(\vec{v})(h_i) = f_s\left(\bigsqcup_{q=1}^{l} \vec{v}(h_{j_q})\right)$$

where h_{j_q} ranges over edges going into h_i's source.

An analogous definition is made for all-paths analyses.

Proposition 2.3.8 *The graph flow function φ is monotone and increases $IN(s)$ and $OUT(s)$.*

Proof. Clearly since f is monotone, φ is also monotone. Now, consider any cycle $t_0, t_1, \ldots, t_n, t_0$ in the plausible graph. Let $g(x)$ represent the action of f on this cycle. Iteration modifies the IN sets using the function $\lambda x.x \sqcup g(x)$, and clearly $x \sqcup g(x) \sqsupset x$. Since f is monotone, then $\mathrm{OUT}(s)$ is also increased. ∎

This function pushes the IN sets of every node through f and puts the resulting value into the OUT set. Note that the first iteration of φ will push a lot of garbage IN sets through f and produce garbage OUT sets; however, soundness spreads through the graph from the correct value at the initial node. Since $\mathcal{D}$ is a cpo, and φ is monotone, we know that φ has a least fixed point, denoted lfp φ.

Definition 2.3.9 *Using the φ operator, we now provide a mathematical definition for $\mathit{fix}_G(f)$:*

$$\mathit{fix}_G(f) = \text{lfp } \varphi$$

Note that the flow function f needs to be multiplicative if we are to prove that the exact analysis modulo f is equal to the result obtained by evaluating the fixed point of f over the graph.

Example 2.3.10 *We consider the example of reaching definitions. For reaching definitions, the cpo is the powerset of the set of pairs* (variable, program point)*. This is a forward analysis, and the merge operator is union. The flow function f, on a definition statement "$s : x =$ value", adds (x, s) to IN(s), and removes any other pairs containing x from IN(s). The initial approximation assigns to all nodes the empty set; this is correct at the start node since no definitions reach it. In this case, we want to compute an approximation of reaching definitions such that every definition that potentially reaches a point p is in the set IN(p).*

2.3.1 Some-path analyses

We wish to show that iterating to the fixed point is correct in an appropriate sense. To do so, we prove the following proposition about some-path flow analyses.

Proposition 2.3.11 *Let f be additive with respect to $\sqcup$. Take any plausible graph G; denote its corresponding plausible set of executions as $\mathcal{P}$. Then*

$$\mathit{fix}_G(f, \sqcup)(s) = \bigsqcup_{e \in \mathcal{P}} OUT_e(s)$$

Proof. We first show that if f is additive, then $\text{fix}_G(f, \sqcup)(s) = \bigsqcup_{p \in \text{paths}(G)} \text{OUT}_p(s)$.

Assume, without loss of generality, that $\perp$ is the initial value. Then,

$$\text{fix}_G(f, \sqcup) = \text{lfp}(\varphi) = \perp \sqcup \varphi(\perp) \sqcup \varphi^2(\perp) \cdots$$

We introduce the notation $\mathrm{OUT}^k = \bot \sqcup \varphi(\bot) \sqcup \cdots \sqcup \varphi^k(\bot)$ to represent the effect of applying the graph flow function φ to the initial approximation k times; $\mathrm{OUT}^k(s)$ is the accumulated approximation at the statement s. Also, $\mathrm{paths}^k(G)$ is the set of paths in G starting at the initial node with length less than k.

The proof is an easy induction; we carry out the induction to point out explicitly where additivity comes in.

For all s, we clearly have that $\mathrm{OUT}^0(s) = \bigsqcup_{p \in \mathrm{paths}^0(G)} \mathrm{OUT}_p(s) = \bot$. Fix k. Assume that $\mathrm{OUT}^{k-1}(s) = \bigsqcup_{p \in \mathrm{paths}^{k-1}(G)} \mathrm{OUT}_p(s)$.

We wish to show that $\mathrm{OUT}^k(s) = \bigsqcup_{p \in \mathrm{paths}^k(G)} \mathrm{OUT}_p(s)$. By definition,

$$\mathrm{OUT}^k(s) = f_s\left(\bigsqcup_{i \in \mathrm{preds}(s)} \mathrm{OUT}^{k-1}(i)\right)$$

Consider the set of paths $\mathfrak{P}$ of length k starting at the initial node and reaching s. We know that, for any path $p \in \mathfrak{P}$, the penultimate node visited is a predecessor of s. Let $\mathcal{I}$ ($\subseteq \mathrm{preds}(s)$) be the set of predecessors of s in paths of $\mathfrak{P}$. We thus have,

$$\bigsqcup_{p \in \mathrm{paths}^k(G)} \mathrm{OUT}_p(s) = \bigsqcup_{p \in \mathrm{paths}^{k-1}(G)} \mathrm{OUT}_p(s) \sqcup \bigsqcup_{i \in \mathcal{I}} f_s\left(\bigsqcup_{p \in \mathrm{paths}^{k-1}(G)} \mathrm{OUT}_p(i)\right)$$

Since we are interested in information at s, $\bigsqcup_{p \in \mathrm{paths}^{k-1}(G)} \mathrm{OUT}_p(s)$ is of the form $f_s(\ell)$, for some element ℓ of the abstraction domain. We are thus computing

$$f_s(\ell) \sqcup \bigsqcup_{i \in \mathcal{I}} f_s\left(\bigsqcup_{p \in \mathrm{paths}^{k-1}(G)} \mathrm{OUT}_p(i)\right) = f_s\left(\ell \sqcup \bigsqcup_{i} \bigsqcup_{p \in \mathrm{paths}^{k-1}(G)} \mathrm{OUT}_p(i)\right)$$

where the equality is a consequence of additivity.

We can further deduce that ℓ is contained in $\bigsqcup_{i \in \mathrm{preds}(s)} \mathrm{OUT}_{\mathrm{paths}^{k-1}}(i)$, because a node must precede s for its contribution to be added as part of $\bigsqcup_{p \in \mathrm{paths}^{k-1}(G)} \mathrm{OUT}_p(s)$. Furthermore, paths used in the computation of ℓ must have length strictly less than k. Hence,

$$\ell \sqcup \bigsqcup_{p \in \mathrm{paths}^{k-1}} \mathrm{OUT}_p(i) = \bigsqcup_{p \in \mathrm{paths}^{k-1}(G)} \mathrm{OUT}_p(i)$$

That is, ℓ is already contained in the union over all paths of length $k-1$.

The equality

$$\bigsqcup_{p \in \text{paths}^{k-1}(G)} \text{OUT}_p(i) = \bigsqcup_{i \in \text{preds}(s)} \text{OUT}^{k-1}(i)$$

holds because of the inductive hypothesis.

We have thus shown that

$$\bigsqcup_{p \in \text{paths}^{k}(G)} \text{OUT}_p(s) = f_s\left(\bigsqcup_{i \in \text{preds}(s)} \text{OUT}^{k-1}(i)\right) = \text{OUT}^k(s)$$

The "corresponding plausible set of executions" is just the set of paths in the graph, so $\text{fix}_G(f, \sqcup)(s) = \bigsqcup_{e \in \mathcal{P}} \text{OUT}_e(s)$, proving the proposition. ∎

Definition 2.3.12 *Given a flow function f for a some-path analysis, a plausible set of executions $\mathcal{E}$, and a flow analysis $\mathcal{F}$, we say that $\mathcal{F}$* soundly approximates *the exact analysis modulo f if*

$$\forall e \in \mathcal{E}.\ \forall s \in G.\ \mathcal{F}_{in}(s) \sqsupseteq IN_e(s)$$

That is, a sound approximation to a some-path analysis must include all the information arising from at least one possible path; it may include more.

We have thus shown

Proposition 2.3.13 *Iterate a flow function to its fixed point over a plausible graph, taking merge operator union. The resulting flow analysis is a sound approximation of the exact flow analysis.*

Soundness is typically what one sees proved in the literature. However this does not rule out an absurdly conservative analysis. In fact most of the analyses in the literature are the "best one can do" given the usual dataflow assumptions, viz. that every edge in the control-flow graph is taken on some execution sequence. We formalize this concept in the following definition.

Definition 2.3.14 *An approximation $\mathcal{F}$ of a flow analysis $\mathcal{F}_0$ is said to be* tight *with respect to a plausible graph G_E if, whenever IN$(s) \sqsupset p$, then $\exists e_1, e_2, \ldots, e_i \in E$. $\exists n_1, n_2, \ldots, n_i \in \mathbb{N}$. where $(s_{n_j}, n_j) \in e_j$ such that*

$$\bigsqcup_j f(e_j|_{n_j}) \sqsupset p$$

This means that if we claim that s has property p, then there is a set T of plausible executions containing s, such that taking the merge of the result obtained by evaluating f on each $t \in T$ gives an element above p. This does not mean that any of these plausible executions actually get taken on some actual execution – if we knew that we would have exact information. Tightness says that the only imprecision in the approximation comes from either not knowing the exact path, or is an artifact of the coarseness of the abstraction domain. Tightness is usually implicit and one does not emphasize it in the sequential case, but for concurrent programs this is important and worth saying explicitly.

Proposition 2.3.15 *Iterating a flow function to its fixed point, with merge operator union, on a plausible graph G_E gives an approximation which is tight with respect to G_E.*

Proof. Also immediate from proposition 2.3.11. ∎

Reaching definitions (as introduced in the extended example) is a some-path analysis.

2.3.2 All-paths analyses

There is a pleasing duality between some-paths and all-paths analysis.

Definition 2.3.16 *An approximation of an all-paths flow analysis is* sound *with respect to a plausible graph E if IN$(s) \sqsupseteq p$ implies that $\forall e \in E$ containing s_n. $f(e|_{n-1}) \sqsupseteq p$ also.*

Definition 2.3.17 *An approximation of an all-paths flow analysis is* tight *with respect to a plausible graph G_E if $\forall s$. if $s_n \in e$, where e is a plausible execution sequence of E, and $f(e|_{n-1}) \sqsupseteq p$ imply that $IN(s) \sqsupseteq p$ also.*

This definition says that if a fact holds through all plausible execution sequences, it must be identified by the approximation.

It is usually appropriate to assign $\top$ to all non-start nodes for all-paths analysis as an initial approximation; the choice of merge operation ensures that the cpo elements strictly decrease on iteration.

We state a proposition analogous to Proposition 2.3.11, but applicable to all-paths analyses.

Proposition 2.3.18 *Let f be multiplicative. Take any plausible graph G; denote its corresponding plausible set of executions as $\mathcal{P}$.*

$$\mathit{fix}_G(f, \sqcap) = \prod_{e \in \mathcal{P}} f(e)$$

The proof is exactly the same as that for Proposition 2.3.11, but the operators are reversed.

Proposition 2.3.19 *Iterating with intersection on a plausible graph gives a tight and correct approximation of the corresponding flow analysis.*

Proof. Immediate consequence of the proposition; tightness and correctness just say that we compute the result over all plausible executions. ∎

An example of an all-paths analyses is available expressions, where the problem is to find expressions which have certainly been computed, but where none of the arguments have changed.

2.3.3 Concluding remarks on flow analysis

We wrap up the discussion of flow analysis with some general observations about flow analysis.

Duality of some-paths and all-paths analyses In this work, we will present results for all-paths and some-paths analyses separately. However, they are actually quite similar. Consider a case where the underlying abstraction domain is a powerset. Typically, an all-paths analysis seeks to find an underapproximation of properties which hold in a program – we cannot introduce any extraneous elements of the powerset – while a some-paths analysis must overestimate the properties which hold. There are no other conceptual differences between the two cases; they are dual.

Backward flow analyses The flow analyses we have discussed to date have been forward analyses, in that the flow function computes the OUT set given the IN set. These analyses give information about events that have occurred in the past. Some analyses are designed to give information about events in the future. For instance, it can be useful to know if there are any references to a variable in the future – that is, whether or not the variable is live. To get this kind of information, we use a *backward* flow analysis.

The distinguishing feature of a backward analysis is that the flow function computes IN sets given OUT sets, and the initial value is for the end node of the graph, rather than the start node. To approximate such an analysis, we can treat it just like a forward analysis on a control-flow graph with reversed edges.

Non-multiplicative flow functions We have previously assumed that our flow functions are always multiplicative or additive, depending on which type of analysis we are considering. If we lack such an assumption, we still have inequalities showing that the result obtained by iterating the graph flow function are sound. In general, the graph flow function's result will approximate, with some error, the result obtained by evaluating the flow function over all plausible sequences. Precision is lost because, for every node s, the graph flow function combines all results at s after each iteration; when the computation over plausible sequences is taken, combining is done only after the function has been computed over all sequences.

Chapter 3

Generating Concurrent Traces

We now return to the model of concurrency described in section 2.1.1, providing more explicit descriptions (in terms of execution sequences) of the effects of our temporal relations on sets of execution sequences. We will also discuss what happens when the relations are not exactly known; often, we are only provided with approximate temporal relations. Even so, we still need to construct a set of plausible execution sequences.

Our scheme for producing a set of plausible execution sequences is as follows. We describe a set of explicit "arrow relations" between program statements. These relations mirror the information gleaned from the program text. From the arrow relations, we can then construct sets of plausible executions, our approximation to the actual executions.

We emphasize that the causal relations between statements determine the set of plausible execution sequences, not the other way around. They are the fundamental objects under consideration. Causal relations are collected by considering the semantics of various statements in the program text; they are then used to create the set of plausible executions that we will reason about. Having a clearly-defined set of plausible executions will allow proofs in later sections: we lift analyses by acting on the relations present at various statements. In order to justify our liftings, we simply consider the effect of the relations on the set of plausible execution sequences. This allows us to show that we are computing the result over all plausible sequences.

We will introduce explicit arrow relations between statements in section 3.1. These relations put constraints on the interleavings between threads and hence on the set of plausible execution sequences. We describe the plausible execution sequences issuing from concurrent programs with these arrow relations. Subsequently, in section 3.2, we will discuss how some concurrency primitives translate to arrow relations. Finally, we discuss the effect of approximation in section 3.5.

3.1 Notation for propagation

In a sequential program, we construct the set of plausible executions by taking all paths in the control-flow graph. A standard control-flow graph, however, has no provisions for concurrency. In this section, we will describe a number of annotations which we add to the control-flow graph so that it can handle concurrency. These annotations are based on the program text. This section does not cover how the annotations are collected for real languages; that is discussed in Chapter 6. Instead, we discuss various concurrency primitives, from a number of languages.

For now, everything is discussed in an intraprocedural context. Chapter 4.4 will describe the necessary modifications to handle an interprocedural analysis of a concurrent program.

Notation for happens-before We recall that we have defined $x \to y$ to mean that x happens-before y. This is not appropriate for propagating information; usually, we want to check that there exists a path between point x and y, such that the path satisfies some property. To check the property at every point between x and y, we cannot use the transitive closure of the $\twoheadrightarrow$ relation. We must use $\twoheadrightarrow$ itself.

The definition of $\twoheadrightarrow$ stated that $x \twoheadrightarrow y$ means that x temporally precedes, possibly immediately, y. We give some informal semantics for $\twoheadrightarrow$.

Axiom 1 *If $x \twoheadrightarrow y$ holds, then the semantics state that:*

- *whenever y is executed, then in every actual interleaving, x has already executed; and,*
- *There exists a actual interleaving in which x occurs immediately before y.*

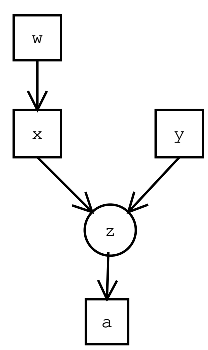
This definition is equivalent to stating that there is no statement z which occurs between x and y on all executions. If there is an actual interleaving putting x immediately before y, then no such statement z could occur. If no statement z must occur between x and y, then x could occur immediately before y.

It is important to recall that $\twoheadrightarrow$ does *not* guarantee that y will execute, and that $x \twoheadrightarrow y$ does not require that x's successors are ordered with respect to events in y's thread. Once a signal has been transmitted, $\twoheadrightarrow$ does not impose any further restrictions on events in the sending thread, regardless of whether or not the signal gets received.

We can use $\twoheadrightarrow$ to represent joining, as in the following example:

```
    T1              T2
w: l = 1729;    y: k = 3;
x: i = 5;          end thread
z: T2.join()
a: j = 2;
```

$\Rightarrow$



In the above example, we have $x \twoheadrightarrow z$ and also $y \twoheadrightarrow z$, but not $w \twoheadrightarrow z$. Also, $w \rightarrow a$ but not $w \twoheadrightarrow a$.

Notation for interleaving Earlier, we introduced the notion of independent instructions: $x \mid\mid y$ means that neither $x \rightarrow y$ nor $y \rightarrow x$ hold. Usually, this means that just before or after x appears in a trace, y could also appear.

A basic concurrency primitive is *mutual exclusion*; languages often provide a notion of "locking", where only one thread can hold a given lock at a time. Other threads attempting to capture the lock must wait until the owner has relinquished the lock. Mutual exclusion does not impose any ordering on program statements.

We can use mutual exclusion to guarantee that after a statement s, an independent statement y does not execute until some lock is freed.

Definition 3.1.1 *If the atomic events x and y are independent, and there exist interleavings where x executes immediately before y and others where y executes immediately before x, then we say that x and y are* proximal. *This is written as $x \bowtie y$.*

In the presence of mutual exclusion or synchronous communication, $\bowtie$ becomes a strict subset of $||$. If $x \bowtie y$ then we certainly have $x \mid\mid y$. Mutual exclusion allows us to have independent statements x and y with no interleavings scrambling them. Synchronous communication imposes additional requirements on the ordering of successors of y, which allows us to remove $\bowtie$ relations from them.

Plausible execution sequences We have described effects of our relations on the set of plausible execution sequences. To recap, if $x \twoheadrightarrow y$, then all appearances of y have a preceding x, and there is a plausible execution where x immediately precedes y. On the other hand, if $x \bowtie y$, then there are plausible sequences containing xy and yx, as well as a sequence where x appears without any adjacent y.

3.2 Various concurrency primitives

A control-flow graph for a concurrent program contains *fork* nodes and causality information. These order the statements. We will now discuss some concurrency primitives and how they translate to our arrow relations. Chapter 6 provides a more detailed description of the mapping between languages and arrows; here, we discuss a selection of primitives, in order to convey a sense of how arrows may be obtained.

Forking A fork node is a special node s in the control-flow graph with two $\twoheadrightarrow$ successors. After a fork node, we say that the flow of execution splits into two threads, each of which are executing concurrently. After any statement a in thread t_0, any other statement z in thread t_1 can execute (so that $a \bowtie z$), and vice-versa. In the absence of further information, all statements in different, concurrently-executing threads are related by $\bowtie$.

Joining A join node is a special node s in the control-flow graph with two $\twoheadrightarrow$ relations into it: $a \twoheadrightarrow s$ and $b \twoheadrightarrow s$, with a, b belonging to different threads. Such a node merges two threads into one. After a join, no $\bowtie$ relations exist between the successors of the join and any predecessors. Joining is equivalent to a rendezvous followed by one of the threads involved in the rendezvous ending. Conversely, a rendezvous can also be modelled by a join followed by the spawning of a new thread.

We can simulate concurrent joins using the previously-introduced arrows. However, it is difficult to recognize a simulated join to use it for improving analysis information. We will thus provide an explicit notation for concurrent joins.

Asynchronous communication If we have additional synchronization information, we can get smaller plausible sets, as we can place fewer $\bowtie$ relations and more $\twoheadrightarrow$ relations.

For instance, in Java, if x is a notify statement and y immediately follows a wait statement, both waiting on the same queue, we can write $x \twoheadrightarrow y$. (Chapter 6 discusses why we must relate the notify with the successor of the wait statement). In that case, we may remove $\bowtie$ relations between predecessors of x and successors of y. Furthermore, we can add $\twoheadrightarrow$ relations between statements.

3.3 Concurrent programs are not like sequential programs

It is tempting to believe that we can treat concurrent programs using the same machinery that we have developed for sequential programs. In particular, one might think that it is possible to add edges representing the possible flow of control to a concurrent thread and treat the resulting graph (now very edge-heavy) as a sequential graph. This idea doesn't work out; below, we elaborate on why this does not work.

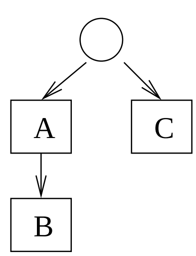
Any statement s can be followed by another statement t if any of the following conditions applies:

- there is a sequential control flow edge from s to t
- $s \twoheadrightarrow t$
- $s \bowtie t$

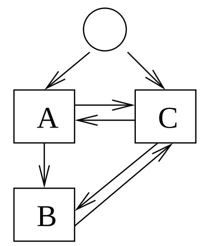
Note that the first possibility is the only one for sequential programs; using it leads to the standard set of plausible executions described for sequential programs.

This set is indeed a plausible set because of the definition of $\bowtie$; it must capture all statements which can execute concurrently with s; adding $\twoheadrightarrow$ and sequential control flow edges complete the picture. However, it is too loose; proving tightness with respect to this plausible set is not proving much at all.

Consider the following situation:



We notice that the following sequential control-flow graph would give exactly the same plausible sequences using the above definition:



Note that there is no actual execution sequence containing the subsequence "BA"; for instance, "BCA" is impossible as an actual execution sequence. Our set of plausible sequences, however, does include "CBCA". In fact, it includes infinite loops when no such loops are possible in any interleaving of the two threads. This leads to grossly inaccurate results – there is no reason for the result of B to be propagated to A.

About our plausible set The above plausible set does not allow us to prove any interesting tightness results. In the concurrent case, we can no longer consider the set of plausible executions to be simply the set of paths in a graph. Instead, we must take our set of plausible executions to be the set of all interleavings of the plausible executions of each concurrent thread. Note that there are two $\forall$ quantifiers in that statement; the outside one is obvious, while the inside one is hidden within the phrase "plausible executions of each concurrent thread". This phrase expands all control-flow choices.

3.4 Summary of concurrency primitives

Table 3.1 summarizes how we obtain the set of plausible executions for a concurrent program, in terms of our primitives. This has been developed in the preceding discussion; we reiterate the rules for creating a set of plausible sequences from typical concurrency primitives. In Chapter 6 we will precisely give the arrows required for each concurrency construct in a number of real languages, and argue that their semantics we have given to the constructs match their actual semantics.

Fork	Distinguished fork node with two $\twoheadrightarrow$ successors. All statements in concurrent threads are $\bowtie$ unless otherwise stated.
Join	Distinguished join node with two $\twoheadrightarrow$ predecessors. No successors of the join have any $\bowtie$ relations with any predecessors of the join.
Mutual Exclusion	Remove $\bowtie$ relations from mutually exclusive statements.
Asynchronous communication	Let y wait for a signal from x. Then $x \twoheadrightarrow y$, and predecessors of x are no longer $\|$ with successors of y, and hence not $\bowtie$.

Table 3.1: From basic concurrency primitives to arrows

3.5 Approximating arrows

In [CS88], it is argued that in the FORTRAN model of post/wait concurrency, it is co-NP hard to compute the "Preserved" sets for parallel FORTRAN programs. As computing these sets are quite similar to computing $\twoheadrightarrow$ information, it is quite likely that we will have to approximate the $\twoheadrightarrow$ and $\bowtie$ information.

We can drop $\twoheadrightarrow$ relations. We need no notion of may-$\twoheadrightarrow$, because $\twoheadrightarrow$ is by nature a *must* relationship; lack of an arrow means that there may be an interleaving between some statements. On the other hand, $\bowtie$ may appear extraneously; the interleaving may occur, but is not bound to occur.

Imagine a situation in which we have a Java wait/notify pair. We have removed the $\bowtie$ relations between the successors of the wait and the predecessors of the notify. If the $\twoheadrightarrow$ relation happened to be omitted, then we would have the negative information – a lack of $\bowtie$s – preventing us from affecting events which are indeed not concurrent, but we would not have positive information: we could not propagate information from the notify to the wait.

Chapter 4

Lifting Sequential Analyses

We will now describe the general lifting at the core of this work. Given a unidirectional sequential flow analysis, we apply a certain transformation, and get a sound analysis for concurrent programs. First, we discuss the types of analyses appropriate for our lifting. For each applicable type of analysis, we will show that our lifting is sound and that, given exact synchronization information, we have the best possible, or tight, analysis result. Finally, we briefly consider the case of interprocedural analyses.

Recall that a forward analysis is one where the flow function describes the OUT set given the value of the IN set and a statement. We start by discussing forward analyses; the situation for backward analyses is quite similar.

4.1 Scope of our liftings

Many flow analyses can be classified according to their merge operators; for instance, a definition made on just one path may reach a program point, while an available expression must be available on all paths to be available at a point. We will also treat points-to analysis, which can simultaneously collect both all-paths and some-paths information.

The information collected will be represented in a complete partial order, which we will denote by $\mathcal{D}$.

Many different abstraction domains are used for flow analysis. Sometimes we track sets of flow objects belonging to finite universes, as in available expressions. Elements of these domains can be represented by finite sets, and the associated analyses are called bitvector analyses. In general, there are often actions on the analysis domain which can be called "kill" and "gen" for each statement. When we have sets of flow objects, kill removes objects from the set, while gen adds objects. Most flow analyses have a flow function of the form

$$\mathrm{out}(s) = \mathrm{in}(s) \setminus \mathrm{kill}(s) \cup \mathrm{gen}(s)$$

The gen and kill sets may be presented in the reverse order for some analyses. For instance, the available expressions analysis generates expressions at a statement and then kills expressions rendered unavailable. On the other hand, in points-to analysis, kills are done first, and then gens are added. To present the results in a uniform way, we can ensure that gen and kill can be commuted by ensuring that the kill set has no intersection with the gen set. This can usually be done by trimming elements from the kill set.

More generally, "kill" and "gen" are actually functions $k_s, g_s : \mathcal{D} \to \mathcal{D}$. For a some-path analysis, kill will tighten the estimate provided by flow analysis, whereas gen loosens the estimate (but is required for correctness). The information ordering is the reverse of the abstraction's poset ordering of the underlying structure. However, in the case of an all-paths analysis, gen tightens the estimate and kill loosens the estimate. In this case, the information ordering coincides with the ordering for the underlying poset. In our analysis, we will insist that the estimate is loosened first, and then tightened.

Definition 4.1.1 *A* kernel operator *is a function K which is idempotent ($K \circ K = K$), monotone ($\ell \sqsubseteq \ell' \Rightarrow K(\ell) \sqsubseteq K(\ell')$) and decreasing ($K(\ell) \sqsubseteq \ell$). A* closure operator *$G$ is idempotent, monotone and increasing ($\ell \sqsubseteq G(\ell)$).*

Axiomatically, we require that:

- k_s is a kernel operator
- g_s is a closure operator

This allows us to rewrite the flow function as follows:

$$\mathrm{out}(s) = (g_s \circ k_s)(\mathrm{in}(s))$$

On a powerset, the standard choices for our functions are $g_s(\ell) = \ell \cup \mathrm{gen}(s)$ and $k_s(\ell) = \ell \setminus \mathrm{kill}(s)$. Note that these choices for k_s and g_s are easily seen to satisfy our axioms.

4.2 All-paths analyses

For a forward all-paths analysis, we have the following prototypical sequential rules:

$$\begin{aligned} \mathrm{in}(s) &= \bigsqcap_{i \in \mathrm{preds(s)}} \mathrm{out}(i) \\ \mathrm{out}(s) &= (g_s \circ k_s)(\mathrm{in}(s)) \end{aligned}$$

Here, a kill just requires one path; a gen must hold on all paths.

We propose the following general rules for a concurrent all-paths analysis on a powerset; we will show that they calculate a sound approximation of the result obtained by evaluating over all plausible sequences. In general, the g_s and k_s functions are used:

$$\begin{aligned} \mathrm{IN(s)} &= (k_{i_1} \circ k_{i_2} \circ \cdots \circ k_{i_n})_{i_1, \ldots, i_n \bowtie s} \left(\bigsqcap_{i \in \mathrm{preds(s)}} \mathrm{OUT}(i) \sqcup \bigsqcup_{i \twoheadrightarrow s} \mathrm{OUT}(i) \right) \\ \mathrm{OUT(s)} &= (k_s \circ g_s)(\mathrm{IN}(s)) \end{aligned}$$

For a bitvector analysis with simple flow functions, we can specialize the above rules using set notation:

$$\begin{aligned} \mathrm{IN(s)} &= \bigcap_{i \in \mathrm{preds(s)}} \mathrm{OUT}(i) \cup \bigcup_{i \twoheadrightarrow s} \mathrm{OUT}(i) \setminus \bigcup_{i \bowtie s} \mathrm{kill}(i) \\ \mathrm{OUT(s)} &= \mathrm{IN}(s) \setminus \mathrm{kill}(s) \cup \mathrm{gen(s)} \end{aligned}$$

The part of the IN rule combining information from predecessors using intersection is the same as for sequential programs. If $x \twoheadrightarrow s$, then at s, we have that on all executions x has already executed. A proximal statement does produce a concurrent kill; this is handled by the $s \bowtie i$ kills.

The above rules use $\sqcup$ to combine information for $\twoheadrightarrow$ concurrent predecessors. Depending on the abstraction, the meaning of $\sqcup$ may not be immediately obvious. It should compute the least upper bound of its arguments, with respect to the information ordering. When the abstraction is just a powerset, then we can indeed take the union; that is the least upper bound. For soundness, any element bounding above all of the arguments of the $\sqcup$ will do. We do this in order to guarantee that $\mathrm{IN}(s) \sqsupseteq \mathrm{OUT}(x)$ for all $x \twoheadrightarrow s$.

We will now argue for the soundness of our rules.

Proposition 4.2.1 *The proposed rules for concurrent all-paths forward analysis compute a sound approximation of the dataflow information.*

Proof. The semantics of the arrow relations allow us to argue for soundness. As we already have soundness for sequential analysis, we need only show that soundness holds in the presence of multiple threads. The changes in the set of plausible executions caused by concurrency are determined by the arrow relations present in the flow graph. We proceed to show that the rules for each type of relation are sound.

Rules for $\bowtie$ are sound If we assert $x \bowtie s$, then the plausible execution sequences include subsequences xs, sx and s. Since we are concerned with the result immediately after s executes, we discard the sx subsequence. Hence we must compute f_x immediately before s executes, combining with the result if f_x never runs. We calculate as follows:

$$\mathrm{IN}(s) = \bigsqcap_{i \in \mathrm{preds}(s)} \mathrm{OUT}(i) \sqcap f_x\left(\bigsqcap_{i \in \mathrm{preds}(s)} \mathrm{OUT}(i)\right)$$

To simplify the notation, let $\ell = \bigsqcap_{i \in \text{preds}(s)} \text{OUT}(i)$. We need to show that $\ell \sqcap k_x \circ g_x(\ell) = k_x(\ell)$. By assumption, we know $\ell \sqsubseteq g_x(\ell)$ and $k_x(\ell) \sqsubseteq \ell$; hence

$$\begin{aligned} & k_x(\ell) \sqsubseteq g_x[k_x(\ell)] \\ \Rightarrow \quad & k_x(\ell) \sqsubseteq \ell \sqcap g_x[k_x(\ell)] \end{aligned}$$

This inequality shows soundness of our rules.

Hence, we need only carry out x's kill at s in order to take into account possible concurrent executions.

Rules for $\twoheadrightarrow$ are sound If $i \twoheadrightarrow s$, then i has already completed, possibly immediately. Any intervening statements are related by $\bowtie$, so their kills have already been accounted for by the above argument.

If s executes, the semantics of $i \twoheadrightarrow s$ guarantee the existence of a plausible execution sequence containing the subsequence is. It is thus clear that the relation $\text{IN}(s) \sqsubseteq \text{OUT}(i)$ holds; from this, we can deduce the soundness of $\text{IN}(s) = \bigsqcap_{i \twoheadrightarrow s} \text{OUT}(i)$.

We assert that it is sound to take the least upper bound, as shown in the rules described above. Consider any ℓ satisfying

$$\bigsqcap_{i \twoheadrightarrow s} \text{OUT}(i) \sqsubseteq \ell \sqsubseteq \bigsqcup_{i \twoheadrightarrow s} \text{OUT}(i)$$

In that case, ℓ is above all of the $\text{OUT}(i)$'s but below some $\text{OUT}(j)$. In particular, ℓ is generated in j's thread, and survives the $\bowtie$ kills corresponding to actions in concurrent threads. Recall that a gen in a thread does not get propagated to concurrent threads; we cannot guarantee that it has definitely occured at any point of the concurrent thread. Since j does definitely occur before s, j asserts ℓ, and no concurrent thread's action has reached j to kill ℓ, it is thus safe to say that ℓ holds at s.

Hence we can soundly let $\text{IN}(s)$ be the least upper bound

$$\text{IN}(s) = \bigsqcup_{i \twoheadrightarrow s} \text{OUT}(i) \sqcup \bigsqcap_{i \in \text{preds}(s)} \text{OUT}(i)$$

This calculation guarantees soundness at $\twoheadrightarrow$ targets; our approximation is contained in the ideal result. ∎

Remark 4.2.2 In order to establish tightness for the $\bowtie$ rule, we must also show that

$$\ell \sqcap g_x[k_x(\ell)] \sqsubseteq k_x(\ell)$$

For powersets and simple kill/gen functions, this is false exactly when some p is killed but then regenerated by a statement x. In this case, we simply insist that $\forall x.\text{kill}(x) \cap \text{gen}(x) = \emptyset$. Given that condition, a simple argument shows that $x \cap ((x \setminus \text{kill}(x)) \cup \text{gen}(x)) = x$.

This condition is more difficult to state if arbitrary kill and gen functions are involved; when needed, we will prove it explicitly for the kill and gen functions at hand. In general, we may need to manipulate the functions so that nothing gets killed and generated in the same statement. If we cannot establish the equality, we only risk a loose approximation – it will be sound. Note that this condition subsumes the one requiring k_x and g_x to commute.

If we can show the technical condition $\ell \sqcap g_x[k_x(\ell)] \sqsubseteq k_x(\ell)$ we have that the rule for $\bowtie$ is equivalent to the result over all plausible sequences, thus giving both soundness and tightness.

Lifting is not tight Figure 4.1 shows that the lifting we have shown so far is not tight. The generation of "r" is inevitable: possible sequences are "ABCD", "ACBD", "ACDB", "CABD", "CADB", and "CDAB"; in each one, a "gen r" occurs. However, the possibly-concurrent "kill r" intervenes and always kills r.

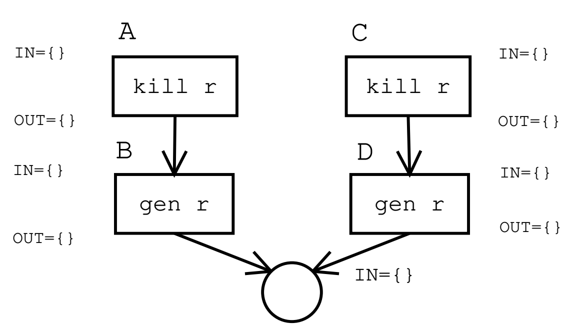


Figure 4.1: Example showing that our lifting is not tight on all-paths

4.2.1 Tightness for bitvector all-paths analyses

Given that our general lifting for all-paths analysis is not tight, we wish to use more sophisticated rules to ensure tightness. Note that we already have proven that our naive lifting is sound; under additional assumptions on the analysis, we can gain tightness.

In this section, we will assume that the abstraction domain is a powerset, and that the flow functions are simple; $g_s(\ell) = \ell \cup \mathrm{gen}(\ell)$, and $k_s(\ell) = \ell \setminus \mathrm{kill}(\ell)$. The presence of statements which are proximal to themselves ($s \bowtie s$) confuses our improved analyses. We do not consider this situation.

Proposition 4.2.3 *Our rule for $\twoheadrightarrow$ is tight as long as $x \bowtie y \Rightarrow gen(x) \cap gen(y) = \emptyset$.*

Proof. Consider some element of the abstraction domain p. For the moment, we are interested in the approximation at statements s such that some number of statements $j_1, j_2, \ldots, j_n$ satisfy $j_i \twoheadrightarrow s$. We have already shown that our rule is sound. We would now like to show that if all plausible sequences declare p at s, then our approximation also declares p at s.

To prove this proposition, we carry out a case analysis. By assumption, a gen of p only occurs at most once in any set of concurrent threads. Let p be generated at statement t. Say p is killed at some statement $i \bowtie t$. Any such kill eliminates p from sets associated with all concurrent statements: this kill may temporally succeed t, so we cannot assert any gens. This is exactly what our $\bowtie$ rule ensures. Since p does not survive on any path, it does not hold at s either. If no kill occurs on any concurrent thread, the sequential rules correctly treat gen and kill of p within the thread where it was generated, and taking unions at $\twoheadrightarrow$ propagates p into s, as desired. ∎

This condition imposes a strong restriction on the types of analyses which can be lifted, if we demand that our liftings be tight. If we carry out the available expressions analysis, making sure that different instances of a given expression are distinct, this condition would be satisfied. An implementation of this sequential analysis is described in a technical report [Lam00]; the analysis considers two instances of `x + y` to be different if they belong to different statements. In general, however, we would like to lift more types of all-paths analyses.

It turns out that if we instead require that the kill sets are disjoint for concurrent statements, we can also prove that our lifting is tight. Although this condition may seem to be more difficult to satisfy, we can use it to provide revised rules which are tight.

Proposition 4.2.4 *The condition $x \bowtie y \Rightarrow kill(x) \cap kill(y) = \emptyset$ guarantees that the lifting we have presented is tight.*

Proof. Consider the prototypical example where s is a statement with two $\twoheadrightarrow$ predecessors (e.g. a `join` node). Statements i and j happen before s and are proximal.

Let some p be killed at a statement i. By assumption, p cannot also be killed at any statement $i' \bowtie i$. We consider the case where $\exists t$ such that $t \bowtie i$ and $p \in \text{gen}(t)$. If that is the case, the concurrent kill at i ensures that p does not survive at any statement concurrent to i – including t – nor at the eventual $\twoheadrightarrow$ successor s.

Thus, the only way that p can survive at an eventual $\twoheadrightarrow$ successor is for some gen of p to happen after any kill of p. Except for $\twoheadrightarrow$ relations, the only way to get such an ordering would be for the gen and the kill to occur on the same thread. This is correctly handled by the sequential rules; our assumption that there are no concurrent kills means that if p survives, it definitely survives unhindered by concurrent kills, and reaches the $\twoheadrightarrow$ successor s. Our rule for $\twoheadrightarrow$ thus gives a tight approximation in this case. ∎

We observe, from the above proposition, that a necessary condition for asserting p at a $\twoheadrightarrow$ node is that, in the presence of any kills of p, an effective gen of p must be on the same thread. We are led to propose additional rules, where we ignore the effects of concurrent threads at each point; we account for concurrent kills only at

$\twoheadrightarrow$ nodes. Since we will take intersections, the concurrent kills affect statements on other nodes.

We first want to track what flow objects potentially get killed in which threads. We first need to associate program statements with threads; we shall separate the statements preceding s into classes. We gather all statements which are $\bowtie$-related to any i_k preceding s in a set I. We then split I into classes $I_1, \ldots, I_n$, according to which thread these nodes belong to. In order to do so, we set $I_k = \{i \in I \mid i \not\bowtie i_k\}$: we put into I_k the elements of I which are not concurrent with i_k. In the presence of mutual exclusion, we are required to add all sequential successors of any nodes in i_k; actions which are parallel but not proximal must be taken into account.

We have already excluded self-concurrent statements; if they were admitted, the I_k set for a self-concurrent statement would be empty, which is undesirable.

Finally, we define HAS-KILL; in any thread, we track which flow objects have been killed. We define it as follows:

$$\text{HAS-KILL}(i_k) = \bigcup_{i \in I_k} \text{kill}(i)$$

For any given thread, these sets add all results from nodes in that thread. A kill need only occur on one thread; we thus use union instead of intersection to combine kill information.

Now we evaluate local sets on all nodes in the I set. These sets are computed using the usual flow analysis rules, except that concurrent effects from statements in I are not considered.

$$\begin{aligned}
\text{IN-LOCAL}(s) &= \left(\bigcap_{i \in \text{preds}(s)} \text{OUT-LOCAL}(i) \cup \bigcup_{i \twoheadrightarrow s} \text{OUT-LOCAL}(i) \right. \\
&\qquad \left. \setminus \bigcup_{i \bowtie s, i \notin I} \text{kill}(i) \right) \cup \ \text{IN}^*(s) \\
\text{OUT-LOCAL}(s) &= f_s\left(\text{IN-LOCAL}(s)\right)
\end{aligned}$$

For an all-paths analysis, initial values for the IN-LOCAL sets should be $\perp$. Note that, when $p \in \text{IN-LOCAL}(s)$, we are not asserting that p holds at s; we can not assert

p as we are not considering the effects of concurrent kills. We use OUT-LOCAL when we have $\twoheadrightarrow$ relations. The following rules will use OUT-LOCAL, albeit indirectly.

$$\begin{aligned}\mathrm{IN(s)} &= \bigcap_{i\in \mathrm{preds(s)}} \mathrm{OUT}(i) \cup \bigcup_{i \twoheadrightarrow s} \mathrm{OUT}(i) \setminus \bigcup_{i \bowtie s} \mathrm{kill}(i) \cup \mathrm{IN}^*(s) \\ \mathrm{OUT(s)} &= \mathrm{IN}(s) \setminus \mathrm{kill}(s) \cup \mathrm{gen(s)}\end{aligned}$$

The calculation of $\mathrm{IN}^*(s)$ is somewhat tricky to describe. Recall that we need only consider elements p which are killed on more than one thread. Let us say that p is killed at least twice. Any threads in which p is not killed cannot influence whether or not p should appear at s: kills of p are interleaved with any potential gen in concurrent threads; only gens that are ordered with respect to the kill can propagate to the $\twoheadrightarrow$. We will take an intersection on OUT-LOCAL sets, so it is in our interest to only include the threads which have kills; other threads can only dilute the information collected, and they are not required for soundness.

We can think about the calculation of $\mathrm{IN}^*(s)$ this way. For each p in the domain, we consider the maximal set I of statements, where $i \in I \Rightarrow (i \twoheadrightarrow s) \wedge (p \in \mathrm{HAS\text{-}KILL}(i))$. We assert $p \in \mathrm{IN}^*(s)$ iff $p \in \bigcap_I \mathrm{OUT\text{-}LOCAL}(i)$.

We first show that at a $\twoheadrightarrow$ node, the intersection of the OUT-LOCAL sets to give IN^* is sound. Then we show that our overall IN sets are tight.

Proposition 4.2.5 *Adding IN*$^*(s)$ *sets at* $\twoheadrightarrow$ *destinations is sound.*

Proof. Let $p \in \mathrm{IN}^*(s)$. Then we have a maximal set I_p for which $p \in \mathrm{HAS\text{-}KILL}(i)$, and p is in the OUT-LOCAL set of each element of I_p. This is enough to ensure that p definitely holds at s: on each thread where p potentially got killed, it gets regenerated. Hence p certainly holds at s. ∎

Proposition 4.2.6 *The proposed IN*(s) *sets are tight for statements with* $\twoheadrightarrow$ *predecessors.*

Proof. We need only consider the case where p is killed on multiple threads because of proposition 4.2.4. Let a statement s have a set S of $\twoheadrightarrow$ predecessors. In some threads, there are kills of p; the set of all predecessors with potential kills of p is called I_p. There are at least 2 elements in I_p. In the presence of concurrency, only the threads in I_p – that is, on which kills of p occur – could regenerate p, as an ordering is needed between the kill and the regeneration.

When considering individual threads, we know that the result of sequential flow analysis is sufficient, as long as we intersect the result over all threads killing p. This accounts for all kills of p. Hence we have reduced the problem of tightness to that for sequential programs, which we have shown earlier. ∎

We did not give an effective algorithm for computing IN^* sets; obviously, it is not practical to iterate over all elements of the abstraction domain. The calculation is actually quite easy for a node which has only two $\twoheadrightarrow$ predecessors i and j; we need only compute

$$\begin{aligned}\mathrm{IN}^*(s) \quad = \quad & (\text{OUT-LOCAL}(i) \cap \text{OUT-LOCAL}(j)) \\ & \cap\ (\text{HAS-KILL}(i) \cap \text{HAS-KILL}(j))\end{aligned}$$

Recall that we need to take the largest subset I_p for which HAS-KILL holds; furthermore, the intersection is only relevant when we are considering more than one thread. Hence the subsets in question must consist of both $\twoheadrightarrow$ predecessors, i and j. It is thus sufficient to take the intersection of the HAS-KILL sets to find the p which are covered by both threads; for these p, we intersect the appropriate OUT-LOCAL sets.

If there are more than two $\twoheadrightarrow$ predecessors for a statement s, we must then consider all subsets I of the set of $\twoheadrightarrow$ predecessors S with at least two members. To ensure that these are the largest possible subsets, we must check that HAS-KILL of all members of I is true, yet HAS-KILL of all nonmembers of I is false. If this holds, we take the intersection of OUT-LOCAL over I. We write concisely:

$$\mathrm{IN}^*(s) = \bigcup_{I \subseteq S} \left(\bigcap_{i \in I} \text{HAS-KILL}(i) \cap \bigcap_{i \in S \setminus I} \overline{\text{HAS-KILL}(i)} \right) \cap \bigcap_{i \in I} \text{OUT-LOCAL}(i)$$

That is, p can be asserted at s if, in the maximal HAS-KILL set of p (which we call I_p), the intersection of the OUT-LOCAL sets of I_p contains p.

4.3 Some-path analyses

We will now treat some-path analyses. In fact, these analyses are quite similar to all-paths analyses; however, a conservative approximation must overestimate the flow sets, instead of underestimating them. As before, we first propose a simple lifting for these analyses, and provide conditions under which this lifting is tight. We then provide additional rules to make our lifting tight.

Consider a typical sequential some-path forward analysis:

$$\begin{aligned} \text{in(s)} &= \bigsqcup_{i \in \text{preds(s)}} \text{out}(i) \\ \text{out(s)} &= (g_s \circ k_s)(\text{in}(s))\text{gen}(s) \end{aligned}$$

Observe that a dataflow object is generated if it appears along just one path, but must be killed on all paths, because the merge operator is union.

In order to lift this analysis to the concurrent case, we need to consider the effect of possible interleavings. Doing so leads to the following lifted rules:

$$\begin{aligned} \text{IN(s)} &= (g_{i_1} \circ g_{i_2} \circ \cdots \circ g_{i_n})_{i_1,\ldots,i_n \bowtie s} \left(\bigsqcup_{i \in \text{preds(s)}} \text{OUT}(i) \sqcup \bigsqcup_{i \twoheadrightarrow s} \text{OUT}(i) \right) \\ \text{OUT(s)} &= (k_s \circ g_s)(\text{IN}(s)) \end{aligned}$$

The same rules can be specialized for bitvector analyses as follows:

$$\begin{aligned} \text{IN(s)} &= \bigcup_{i \in \text{preds(s)}} \text{OUT}(i) \cup \bigcap_{i \twoheadrightarrow s} \text{OUT}(i) \cup \bigcup_{i \bowtie s} \text{gen}(i) \\ \text{OUT(s)} &= (\text{IN}(s) \setminus \text{kill}(s)) \cup \text{gen}(s) \end{aligned}$$

As expected, we produce concurrent gen's at every point, reflecting possible executions. A twist is the use of the intersection operator – that is, taking the greatest lower bound – for combining information from concurrent threads. We argue for the soundness of these rules. In general, they are not tight.

Proposition 4.3.1 *The proposed rules for lifting a some-paths concurrent analysis are sound.*

Proof. Once again, we proceed by offering an argument based on the structure of the plausible execution sequences, which is completely determined by the arrow relations asserted.

Treatment of $\bowtie$ is sound When the relation $x \bowtie s$ holds, plausible executions contain s and xs. Once again, we set up $\ell = \bigsqcap_{i \in \text{preds}(s)} \text{OUT}(i)$. At s we evaluate,

$$\text{IN}(s) = \ell \sqcup f_x(\ell)$$

The bound $\ell \sqcup g_x(k_x(\ell)) \sqsubseteq g_x(\ell)$ is clear; this guarantees soundness.

Hence, the correct treatment for $\bowtie$-related statements is to add the gen set.

Treatment of $\twoheadrightarrow$ is sound Let s_0 and s_1 have a common join successor t (so that $s_0 \twoheadrightarrow t, s_1 \twoheadrightarrow t$). Clearly $\text{OUT}(s_0) \sqcup \text{OUT}(s_1)$ is a sound approximation of $\text{IN}(t)$, but we want to prove that it is sound to take $\text{OUT}(s_0) \sqcap \text{OUT}(s_1)$.

We reuse the argument from the all-paths case. Consider any element ℓ of the domain between the intersection and the union:

$$\bigsqcap_{i \twoheadrightarrow s} \text{OUT}(i) \leq \ell \leq \bigsqcup_{i \twoheadrightarrow s} \text{OUT}(i)$$

We claim that any such ℓ is a sound estimate for $\text{IN}(s)$.

Consider some such ℓ. It is below every one of the $\text{OUT}(i)$'s and above some $\text{OUT}(j)$. This corresponds to some kill in j's thread which survives concurrent gens to reach the join node. Since kills do not propagate to concurrent threads, taking the union is clearly suboptimal. Now, j definitely occurs before s, j asserts something less than ℓ, and no concurrent thread gens something above ℓ. Hence we can assert that the exact answer at s is at most ℓ.

Since this applies for all such ℓ, we can soundly declare

$$\mathrm{IN}(s) = \bigsqcap_{i \twoheadrightarrow s} \mathrm{OUT}(i) \sqcup \bigsqcup_{i \in \mathrm{preds}(i)} \mathrm{OUT}(i)$$

∎

Remark 4.3.2 When we proved that the rule for $\bowtie$ was sound, we established the bound,

$$\ell \sqcup g_x(k_x(\ell)) \sqsubseteq g_x(\ell)$$

The reverse bound follows either if we establish the technical condition ensuring that gen $\cap$ kill $= \emptyset$, or by a direct proof for some specific analysis. If the reverse bound is established, then our rule for $\bowtie$ is also tight.

Example of a non-tight lifting Consider the variant of constant propagation where the merge operator combines the possible constants on each branch. For instance, if $\{(x, 5)\}$ is propagated on one branch, and $\{(x, 3)\}$ is propagated on another branch, then at the merge point, $\{(x, 5), (x, 3)\}$ would be asserted.

The fact that our lifting is not tight is illustrated by the program fragment in figure 4.2. Inspection of the fragment shows that we need not assert $(x, 1)$ at the join node, but it is not eliminated by the intersection, because it gets regenerated on both threads.

4.3.1 Tightness for bitvector some-paths analyses

We now restrict our attention to analyses which operate on powersets. Once again, we assume that the flow functions are of the form $g_s(\ell) = \ell \cup \mathrm{gen}(s)$ and $k_s(\ell) = \ell \backslash \mathrm{kill}(s)$. We also do not consider self-parallel statements.

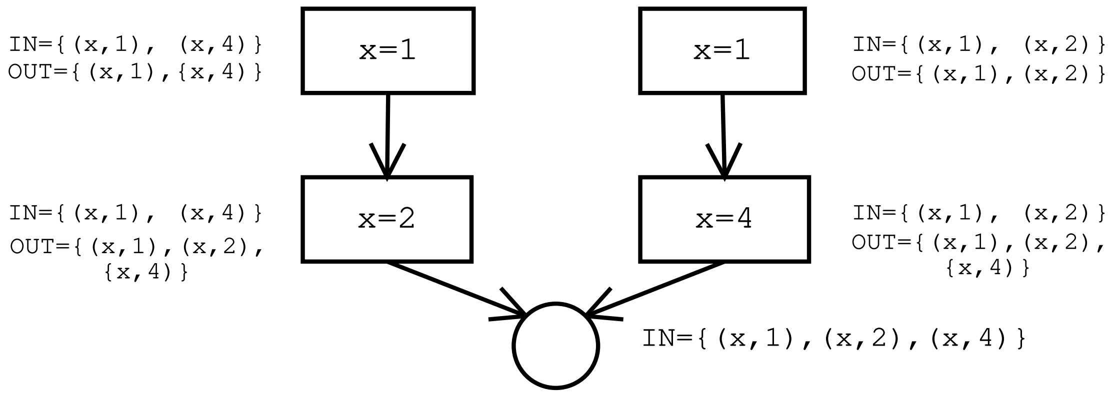


Figure 4.2: Example showing that our naive lifting is not tight

As in the all-paths case, we claim that if flow objects also carry information distinguishing identical gens in different threads, then the situation in the figure cannot arise, and tightness would hold. Similarly, if kills are not shared between threads, tightness also holds.

Proposition 4.3.3 *Our rule for $\twoheadrightarrow$ is tight when $x \bowtie y \Rightarrow kill(x) \cap kill(y) = \emptyset$.*

Proof. Let some element p be killed at a statement t; by assumption, it can be killed nowhere else in the set of concurrent statements. If p is generated at some concurrent $i \bowtie t$, then the kill of p does not have any effect; this is handled by the $\bowtie$ rules. Otherwise, the usual sequential rules correctly handle the kill of p on t's thread with respect to interfering gens, ensuring the correct result holds at the predecessor to s. Taking an intersection at s propagates the kill of p to s exactly when all plausible sequences kill p. ∎

Proposition 4.3.4 *Our rule for $\twoheadrightarrow$ is tight when $x \bowtie y \Rightarrow gen(x) \cap gen(y) = \emptyset$.*

Proof. This proof mirrors the one from the all-paths case when we insisted that kill sets be disjoint on concurrent nodes. Let p be generated at some statement t. We assume that there are no gens of p on any concurrent threads. Any kills of p concurrent to t have no effect, because they have no ordering with respect to the gen at t; only kills in t's thread could cause p to be removed from flow sets. Such kills are correctly handled by the sequential rules for t's thread: a kill takes effect at s exactly when there are no concurrent gens of p, as long as the kill of p is propagated to s by the sequential rules. ∎

Since there is a counterexample showing that our simple lifting is not tight, we will have to improve the rules in order to get a tight lifting.

This time, it is necessary to track only threads on which gens occur; these are the ones that can affect the result at a $\twoheadrightarrow$ node. To do so, we reuse the I_k sets introduced for all-paths analysis in section 4.2.1, which track sets of statements belonging to various threads. However, now we compute whether or not a thread has any gen.

$$\text{HAS-GEN}(i_k) = \bigcup_{i \in I_k} \text{gen}(i)$$

We also compute local sets, ignoring the effect of concurrent threads, for all nodes in I. Initial values for each of the IN-LOCAL sets should be $\top$: we assume that there are no kills in the past.

$$\begin{aligned}
\text{IN-LOCAL}(s) &= \left(\bigcup_{i \in \text{preds}(s)} \text{OUT-LOCAL}(s) \cup \bigcap_{i \twoheadrightarrow s} \text{OUT-LOCAL}(i) \right. \\
&\qquad \left. \cup \bigcup_{i \bowtie s, i \notin I} \text{gen}(i) \right) \cap \ \text{IN}^*(s) \\
\text{OUT-LOCAL}(s) &= f_s(\text{IN-LOCAL}(s))
\end{aligned}$$

These allow us to construct IN^* sets. For each p in the domain, we consider the maximal set I_p of CFG nodes on which HAS-GEN(p) holds. Then, if s has multiple $\twoheadrightarrow$ predecessors, we say $p \in \text{IN}^*(s)$ iff $p \in \bigcap_{i \in I_p} \text{OUT-LOCAL}(i)$. Hence the IN^* set tracks statements which must be let through; in particular, the elements which have been killed on any thread and not regenerated are not let through. A formula for $\text{IN}^*(s)$ is

$$\text{IN}^*(s) = \bigcup_{I \subseteq S} \left(\bigcap_{i \in I} \text{HAS-GEN}(i) \cap \bigcap_{i \notin I} \overline{\text{HAS-GEN}(i)} \right) \cap \bigcap_{i \in I} \text{OUT-LOCAL}(i)$$

We can propose the following improved rules for lifting some-path analyses.

$$
\begin{aligned}
\mathrm{IN}(s) &= \left(\bigcup_{i \in \mathrm{preds}(s)} \mathrm{OUT}(i) \cup \bigcap_{i \twoheadrightarrow s} \mathrm{OUT}(i) \cup \bigcup_{i \bowtie s} \mathrm{gen}(i) \right) \cap\ \mathrm{IN}^*(s) \\
\mathrm{OUT}(s) &= (\mathrm{IN}(s) \setminus \mathrm{kill}(s)) \cup \mathrm{gen}(s)
\end{aligned}
$$

Proposition 4.3.5 *For all s, the IN$^*(s)$ sets contain the result obtained by evaluating the flow function over all plausible sequences and are hence sound; we may intersect them with the naive IN(s) sets.*

Proof. Let s be a statement with multiple $\twoheadrightarrow$ predecessors. Assume $p \notin \mathrm{IN}^*(s)$. We will show that there is no plausible execution reaching s for which the evaluation of f contains p. Now, we have the set HAS-GEN of predecessors with a gen of p. Our assumption $p \notin \mathrm{IN}^*(s)$ implies that $p \notin \text{OUT-LOCAL}(i)$ on each thread where p might have been generated: there is a kill of p whenever there is a gen. Note that other threads, not being in HAS-GEN, will not contribute to deciding whether or not p holds at s.

We discuss the case where p is in the IN set of some predecessor of a node in I. If there happen to be no gens of p at all in I, then I_p is empty. By convention, the intersection $\bigcap_{i \in I_p} \text{OUT-LOCAL}(i)$ contains p (and every other element of the domain) if I_p is empty. When there are some gens of p in I, then we have the above situation; the presence of p in the IN set of a predecessor is irrelevant.

In summary, no plausible execution sequence will omit any information at s; the IN* set is sound. ∎

Proposition 4.3.6 *The proposed rule for the IN sets, including the intersection with IN$^*(s)$, is tight.*

Proof. We now need to show that any $p \in \mathrm{IN}(s)$ must be there: for all plausible executions, there is some gen of p and a kill-free path from the gen to s.

As in the all-paths case, we have narrowed the question down to proving tightness for the case in which some p is generated on multiple threads. Assume, without loss of generality, that p is generated on at least two threads. The set HAS-GEN tracks all threads on which p is generated. If $p \in \mathrm{IN}^*(s)$, then on some thread where p is generated, it is not killed. If this gen is run after all interleavings where p is killed, we exhibit a plausible execution where p holds at s. ∎

Backwards analyses Backwards analyses follow the same basic rules, with directions reversed. For instance, sequential rules for an all-paths analysis are:

$$\begin{aligned} \mathrm{out}(s) &= \bigsqcap_{i \in \mathrm{succs}(s)} \mathrm{in}(s) \\ \mathrm{in}(s) &= f_s(\mathrm{out}(s)) \end{aligned}$$

We can view a backwards analysis as a forward analysis on a control-flow graph with reversed arrows. In terms of execution sequences, a backwards analysis does not consider heads of sequences, but instead tails of sequences; it seeks to discover properties that will hold in the future, rather than properties about the past.

We provide the rules for a naive lifting for the backwards all-paths analysis. We omit rules for the tight lifting and proofs of any sort; these can be deduced easily from the treatment of the forward lifting.

$$\begin{aligned} \mathrm{OUT}(s) &= \bigsqcap_{i \in \mathrm{succs}(s)} \mathrm{OUT}(s) \sqcup \bigsqcup_{i \bowtie s} \mathrm{gen}(i) \\ \mathrm{IN}(s) &= f_s(\mathrm{OUT}(s)) \end{aligned}$$

4.4 Interprocedural analysis for concurrent programs

It is often quite important to analyse programs interprocedurally; analysing just one method at a time forces very conservative assumptions to be made. This is especially true for concurrent programs. For instance, in Java, starting a new thread is an interprocedural operation: the new thread starts executing the `run()` method of some object, instead of continuing the method which was running previously.

An interprocedural analysis on a concurrent program has significantly more interference from other procedures than one on a sequential program. In a sequential program, the program points where procedures are called are readily identifiable, even if the target might not be known at compile-time. A concurrent program, on the other hand, has possible interleavings at every statement, although the interleavings are not equivalent to procedure calls: no mapping of arguments is made, for instance.

We believe that many of the fundamental problems in lifting dataflow analyses arise in the intraprocedural case, and that proposing intraprocedural liftings illustrate the most important principles involved in analysis of concurrent programs. Nevertheless, attempting to lift interprocedural analyses is still quite complicated, and we do not treat this situation in this work.

Chapter 5

Examples of Analyses and Their Liftings

The claim that we make in this work is that it is fairly straightforward to consider a standard sequential program analysis and provide the corresponding analysis for concurrent programs. This will now be demonstrated: we lift several sequential analyses to the concurrent case.

5.1 Available expressions

A standard program optimization is common subexpression elimination, where redundant computations of an expression are suppressed. To carry out common subexpression elimination, the results of the *available expressions* analysis are required. The sequential version of this analysis has been implemented and the practical behaviour of the common subexpression elimination optimization on Java programs has been investigated in a technical report [Lam00]. We will now describe this analysis and its lifting.

This analysis operates on the universe of expressions. Elements in this universe can be represented by bitvectors, so the situation is particularly simple here. In order to carry out common subexpression elimination, we require that distinct instances of an expression (for instance, $x + y$ may occur at two program points) be treated as different expressions. With this requirement, our lifting is guaranteed to be tight because of proposition 4.2.3: no two statements generate the same expression.

An expression e of the form `x op y` is *available* at program point p if all paths from some evaluation of e to p is free of redefinitions of x and y. (Of course, we also allow unary and tertiary operators.) The analysis is forward; the merge operator is $\cap$ for sequential merges.

For this analysis, we define the sets at program point s:

$$\begin{aligned} \text{gen(s)} &= \text{expressions evaluated in } s \\ \text{kill(s)} &= \text{expressions containing variables defined in } s \end{aligned}$$

To meet the technical condition that, for all s, $\text{kill}(s) \cap \text{gen}(s) = \emptyset$, at s we do not kill expressions that are evaluated in a statement s.

Because $\bigcup_{i \bowtie s} \text{kill}(i)$ is well-defined, the lifting is particularly simple; all bitvector analyses share this property. A straightforward application of the naive lifting gives the following rules.

$$\begin{aligned} \text{IN}(s) &= \bigcup_{i \in \text{preds}(s)} \text{OUT}(i) \cup \bigcap_{i \twoheadrightarrow s} \text{OUT}(i) \setminus \bigcup_{i \bowtie s} \text{kill}(i) \\ \text{OUT}(s) &= \text{IN}(s) \setminus \text{kill}(s) \cup \text{gen}(s) \end{aligned}$$

We close the example by working through a number of simple flow graphs. First, we illustrate available expressions for a sequential program in Figure 5.1; it is fairly simple but illustrates how information is flowed around a sequential CFG. In these figures, the sets illustrated are always OUT sets.

We also illustrate the analysis on the concurrent program in figure 5.1. Assume that we have matched the `notify()/wait()` pair; this allows us to calculate the arrow relations. We point out that $d = 2$ prevents $c + d$ from becoming available in the concurrent thread. On the other hand, synchronization ensures that $x + 3$ becomes

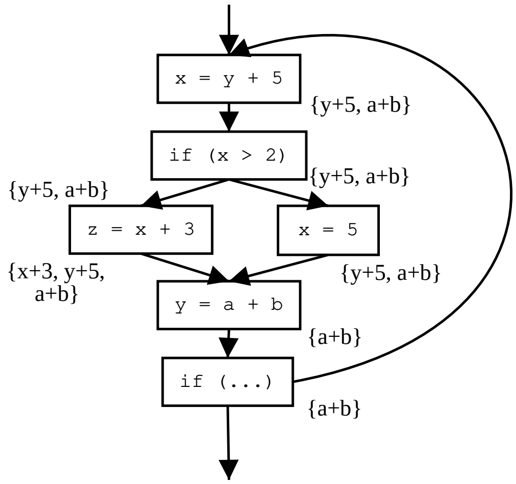


Figure 5.1: Available expressions on a sequential graph

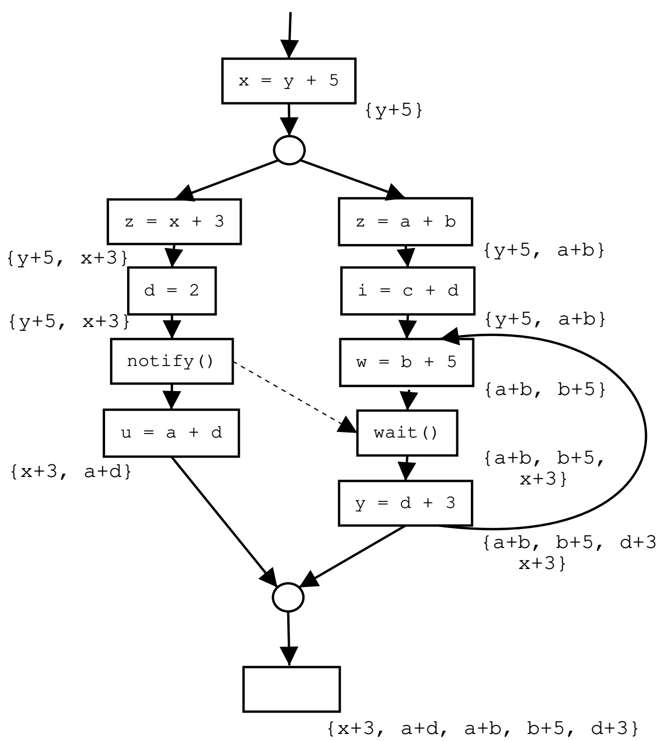


Figure 5.2: Available expressions on a concurrent graph

available on the other thread, because it is true that the computation has certainly executed before anything after the `wait()`. Finally, after the concurrent join, the sets on both paths are combined using set union.

5.2 Live variables analysis

Our next example will treat the *live variables* problem. A variable is *live* at program point s if its value can be used in the future on some execution. There are obvious applications to register allocation; this analysis is also useful in loop optimizations, for instance to remove loop invariants.

The live variables analysis operates on the universe of variables in a program. Once again, this is a bitvector analysis, so we have the more precise results from Chapter 4 at our disposal.

At a program point s, a variable v is *live* if its value has a use in some path from s to the exit node. This is a backwards some-paths analysis; its merge operator is $\cup$. We define the gen and kill sets.

$$\begin{aligned} \text{gen}(s) &= \text{variables which are used in } s \\ \text{kill}(s) &= \text{variables which are defined in } s \end{aligned}$$

Note that there may well be gens or kills which occur at more than one statement, so we must apply the refined lifting.

Given this data, we are able to apply our lifting to obtain a concurrent analysis. This requires no additional effort: in the sequel, we simply restate flow equations from Chapter 4. They are minimally altered because live variables is a backwards analysis.

Now, our lifting relies on collecting HAS-GEN sets. As we have a backwards analysis, these sets are collected for each node s with multiple $\twoheadrightarrow$ successors (for instance, fork and notify nodes). We collect sets of I_k statements for each concurrent successor i_k of s. This allows us to set

$$\text{HAS-GEN}(i_k) = \bigcup_{i \in I_k} \text{gen}(i)$$

Next, we compute local sets for all nodes in I.

$$\begin{aligned}
\text{OUT-LOCAL}(s) &= \left(\bigcup_{i \in \text{preds}(s)} \text{IN-LOCAL}(s) \cup \bigcap_{i \twoheadrightarrow s} \text{IN-LOCAL}(i) \right. \\
&\qquad \left. \cup \bigcup_{i \bowtie s, i \notin I} \text{gen}(i) \right) \cap \ \text{OUT}^*(s) \\
\text{IN-LOCAL}(s) &= f_s(\text{OUT-LOCAL}(s))
\end{aligned}$$

From the IN-LOCAL sets, we construct the OUT* sets for statements s with sets of multiple $\twoheadrightarrow$ successors S as follows:

$$\text{OUT}^*(s) = \bigcup_{I \subseteq S} \left(\bigcap_{i \in I} \text{HAS-GEN}(i) \cap \bigcap_{i \notin I} \overline{\text{HAS-GEN}(i)} \right) \cap \bigcap_{i \in I} \text{IN-LOCAL}(i)$$

In all, we have the rules

$$\begin{aligned}
\text{OUT}(s) &= \left(\bigcup_{i \in \text{succs}(s)} \text{IN}(i) \cup \bigcap_{i \twoheadrightarrow s} \text{OUT}(i) \cup \bigcup_{i \bowtie s} \text{gen}(i) \right) \cap \text{OUT}^*(s) \\
\text{IN}(s) &= (\text{OUT}(s) \setminus \text{kill}(s)) \cup \text{gen}(s)
\end{aligned}$$

We conclude the discussion of live variables analysis with two examples of the analysis on concurrent programs.

The first example, in figure 5.2, demonstrates the use of the OUT* sets for the variable y. Also note that z is killed only on the left thread; since there is no gen on the right thread, z does not survive past the fork node.

In the second example, shown in figure 5.2, we nest concurrency constructs, and include a wait/notify construct inside the inner set of threads. Space does not permit the inclusion of the LOCAL sets; we do illustrate the OUT* sets, though. Notice in particular that the x on the internal set of threads gets killed by the definition $x = 3$. Also, note that the gen of x on the left-hand thread must be included on the rightmost thread, even if it is eventually killed at concurrent merge points.

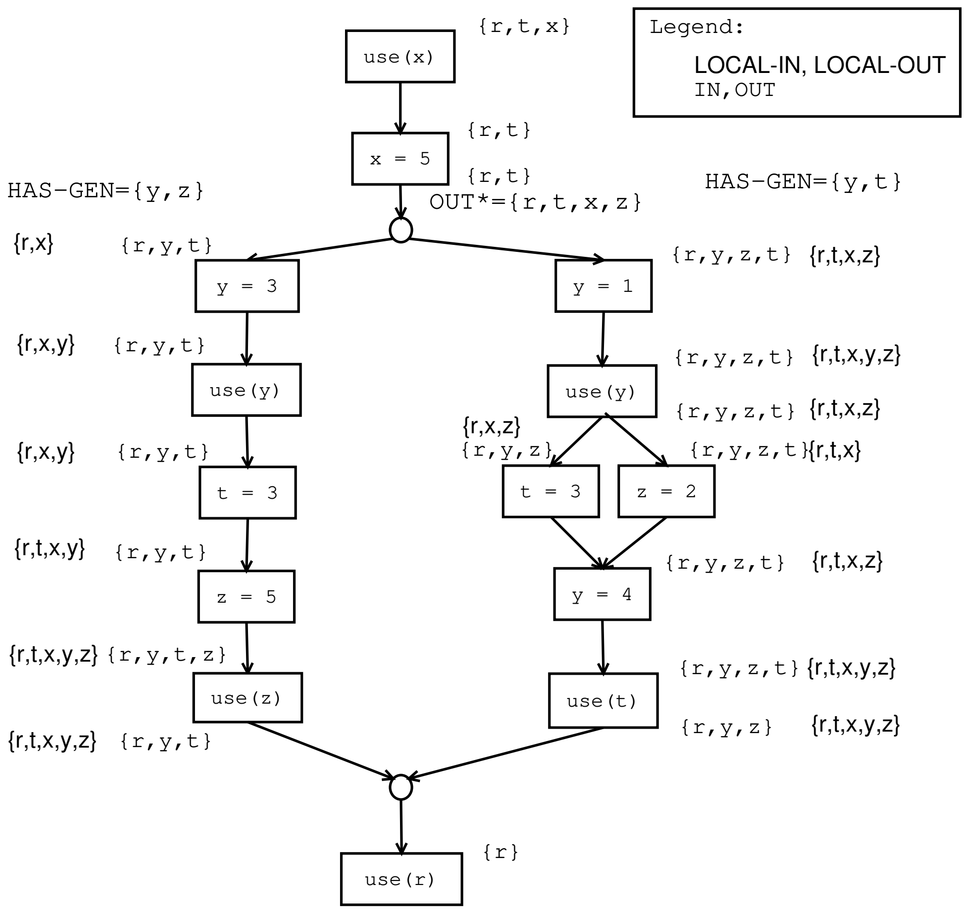


Figure 5.3: Live variables analysis on a simple concurrent graph

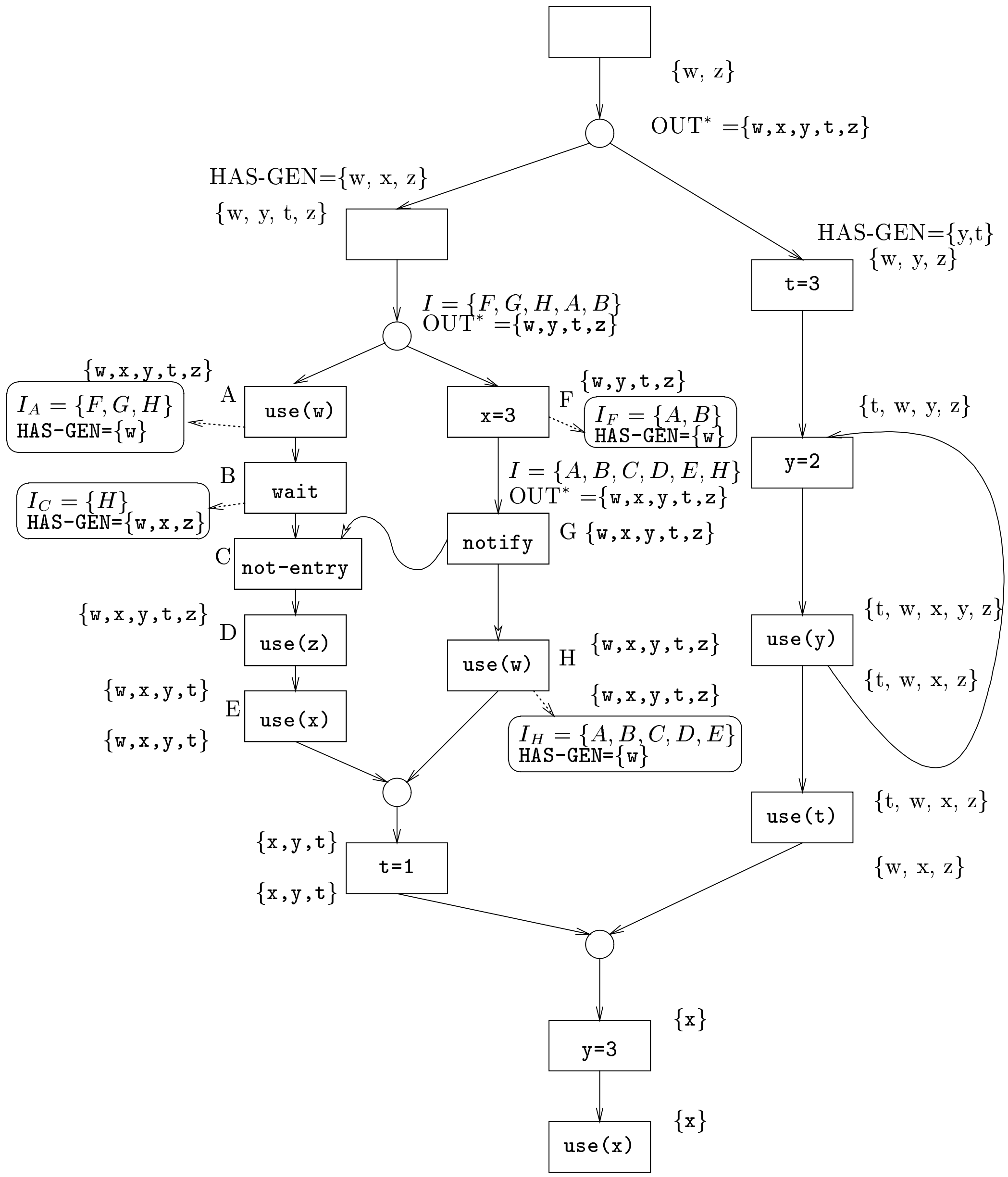


Figure 5.4: Live variables analysis on a complicated concurrent graph

5.3 Generalized constant propagation

Some analyses use more interesting abstractions than the ones used in available expressions and live variables. For instance, *generalized constant propagation* propagates ranges of values for variables. We discuss its lifting to a concurrent analysis.

As presented in [VCH96], generalized constant propagation is applicable to C programs; it was implemented in the McCAT C compiler framework. As such, it assumes that the program under consideration is structured: that is, it assumes that the program is presented in Abstract Syntax Tree form. However, it is fairly straightforward to convert this analysis to one that operates on a control-flow graph.

The underlying abstraction domain is that of closed scalar intervals: we consider ranges $[a, b]$ with $\perp = []$ and $\top = [-\infty, \infty]$, where ∞ is the largest machine representable scalar. We can define an ordering and a meet operator on this domain. The relation $[a, b] \sqsubseteq [c, d]$ holds if $a \geq c$ and $b \leq d$. We define the meet by $[a, b] \vee [c, d] = [\min(a, c), \max(b, c)]$.

One special feature of generalized constant propagation is that it is a *branched* analysis. At a conditional statement (e.g. `if (x < 0) then goto y`), we can propagate additional information to y: namely, that $x < 0$ holds. This means that we actually have several distinct OUT sets from a statement.

To describe the actual analysis, we now present a number of the sequential flow functions for a number of different statement types. Working out the other flow functions is a fairly simple exercise.

In this all-paths analysis, the k_s function loosens the estimate and the g_s function tightens it. As ranges, $\ell \sqsubseteq k_s(\ell)$ and $g_s(\ell) \sqsubseteq \ell$; this is required to satisfy our assumption that k_s and g_s are kernel and closure operators.

We briefly discuss the technical proviso guaranteeing $\ell \sqcap k_s(g_s(\ell)) = k_s(\ell)$. In our case, k_s will always widen the range of ℓ, computing the meet of ℓ and what s requires; g_s then narrows the information. If k_s always combines ℓ with some range, while g_s returns the range itself (it is a constant function: the output of any given g_s

function does not depend on ℓ), we can see that $\ell \sqcap g_s(k_s(\ell)) = k_s(\ell)$. For any ℓ_0, the intersection $\ell \sqcap g_s(\ell_0)$ is exactly k_s: we always require that k_s computes the meet of ℓ and the range generated by g_s.

The simplest example is the statement

$$s : x = \texttt{c}$$

for some constant c. The associated functions are $k_s(\ell) = (\ell \vee t : [\texttt{c}, \texttt{c}])$: we ensure that ℓ contains c; and $g_s(\ell) = t : [\texttt{c}, \texttt{c}]$, strictly improving the estimate.

For a statement s of the form

$$s : x = y$$

the kill function must ensure that x's range is widened to contain the range for y in the IN set for s: $k_s(\ell) = x : \texttt{RangeOf}(x, \text{IN}(s)) \vee \ell$, while the gen function narrows x's range to be exactly the range given by y: $g_s(\ell) = x : \texttt{RangeOf}(x, \text{IN}(s))$. Again, the gen function strictly improves the estimate.

The original paper also presented rules for use with C-style pointer operations. Even though we do not consider models of languages with C-style pointers in our discussion of real languages, it is still meaningful to discuss the liftings of these rules. These rules require points-to analysis. Points-to analysis is a more complicated analysis for sequential programs; we will not discuss it in this work, although there are no conceptual barriers to lifting it.

In the case of a statement

$$s : x = *y$$

we want to merge the ranges for every possible value that y might point to (that is, Dereference(y)); this is the new range for x. This forces

$$\begin{aligned} k_s(\ell) &= \ell \vee x : \text{MergeRanges}(\text{RangesOf}(\text{Dereference}(y))) \\ g_s(\ell) &= x : \text{MergeRanges}(\text{RangesOf}(\text{Dereference}(y))) \end{aligned}$$

narrowing the estimate removing the influence of the original ℓ.

We also must consider statements of the form

$$s : *x = y$$

In that case, we may have definite or possible points-to information about x. If we know that x definitely points to some location z, we can treat the statement exactly as if it read $z = y$.

On the other hand, we may only have possible information about x: it may point to either d, e or f. In that case, there is uncertainty about the actual effect of this statement; its effect must be estimated. This uncertainty is actually quite similar to the uncertainty encountered when we have proximal statements in the execution. For each z possibly-pointed to by x, we set $k_s(\ell) = \ell \vee z : \text{RangeOf}(y)$, merging the old value for z (from ℓ) with the known range for y. We cannot further tighten the estimate, so $g_s(\ell) = \ell$.

A statement can involve an add expression, like

$$s : x = y + z$$

We need to estimate its effect on the flow sets. Given two incoming ranges $y : [a, b]$ and $z : [c, d]$, we abstractly add them as follows: the lower bound e is $-\infty$ if $a + c \leq -\infty$; otherwise it is $a + c$. Similarly, the upper bound f is ∞ if $b + d \geq \infty$; otherwise it is $b + d$. The kill function merges the information: $k_s(\ell) = \ell \vee [e, f]$ and the gen function removes the old information: $g_s(\ell) = [e, f]$.

Conditionals and looping The original paper discussed the effect of conditionals and looping; a structured analysis has explicit conditional and loop instructions to consider. Our analysis must take a different approach.

We discuss in further detail the branched OUT sets at conditional branches. We earlier alluded to the existence of multiple OUT sets for a conditional branch node. More precisely, we associate to each outgoing control-flow edge an OUT set. One edge is taken if the test condition is true; the other edge is taken if it is false.

The condition allows us to split the IN set into two parts: one consistent with the condition, and one inconsistent with it. For instance, $x > 0$ would split the range $x : [-5, 5]$ into the consistent range $x : [0, 5]$ and the inconsistent range $x : [-5, 0]$. Clearly, we propagate the consistent ranges on the taken edge, and the inconsistent ranges on the edge not taken.

Loops are handled by the fixed point computation. However, a notable feature of the paper is artificial stepping-up of ranges, in order to guarantee reasonable convergence times. An intelligent conversion of the stepping-up from a structured analysis to one for CFGs is beyond the scope of this work. A simplified stepping-up would count the number of times a CFG node is iterated; if this number exceeds a certain arbitrary limit, the ranges would be increased, say by widening to $[-\infty, \infty]$. This is sound but far from tight.

Lifting generalized constant propagation As this analysis does not operate on a powerset, our naive lifting is not guaranteed to be tight. However, it will be sound.

Applying our rules to this analysis, we see that concurrent kills must be applied:

$$\mathrm{IN}(s) = (k_{i_1} \circ \cdots \circ k_{i_n})_{i_j \bowtie s} \left(\bigvee_{i \in \mathrm{preds}(s)} \mathrm{OUT}(i) \vee \bigwedge_{i \twoheadrightarrow s} \mathrm{OUT}(i) \right)$$

We proceed to illustrate two examples of generalized constant propagation on concurrent programs.

The first example, in figure 5.3, demonstrates generalized constant propagation on a simple example. We omit variables with ranges $[-\infty, \infty]$. Note that the `if` statement in the left thread can generate additional constraints, while the one on the right cannot; concurrent gens interfere. Also, we point out the use of $\wedge$ to merge the sets at the concurrent join node.

In our second example, shown in figure 5.3, we illustrate how our simple lifting is not tight. The lifted analysis works well for the at the first join node; intuitively, we see that there is only one gen per thread for x. However, the context insensitivity

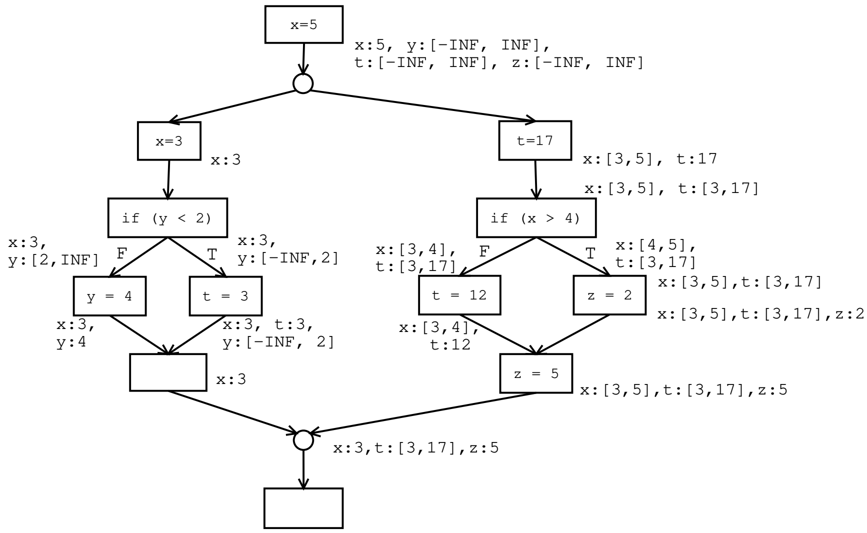


Figure 5.5: Generalized constant propagation on a simple concurrent graph

of our lifting prevents us from getting exact information about the second pair of threads. In particular, from looking at the program we see that an exact analysis would say that x must be in the range $[17, 23]$ after the second concurrent join.

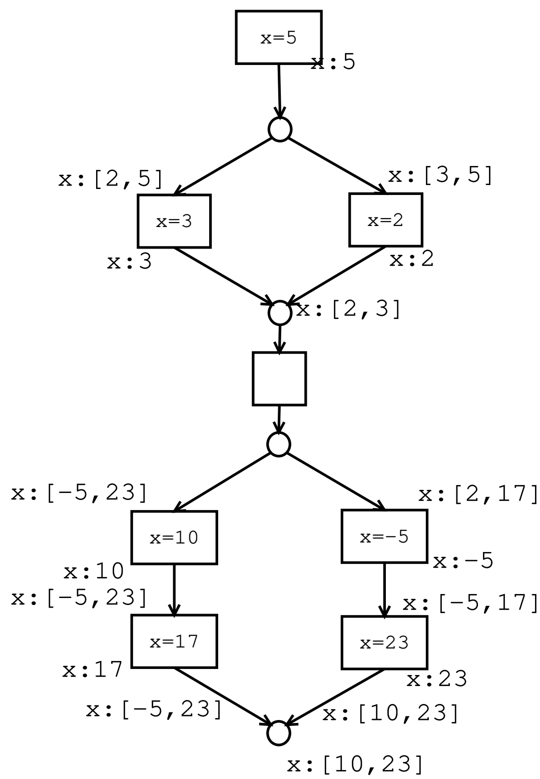


Figure 5.6: Lifting of generalized constant propagation is not tight

Chapter 6

Detecting Concurrency Relations in Real Languages

Our concurrency model abstracts away from concurrency features in actual languages, replacing them with $\twoheadrightarrow$ and $\bowtie$ relations. We will now convert constructs which appear in real languages into the above-mentioned relations. In this work, we have chosen to handle Java, Ada and CML, representing three different concurrency models; adding support for other concurrency models would be fairly straightforward.

6.1 Java

For many reasons, the Java programming language has recently become a popular language for programming languages researchers to study. Optimizing compiler frameworks exist for Java; one of them is the Soot framework [VR00]. Furthermore, Java has support for concurrency built into the language specification [GJS00] itself. Hence, we discuss the mapping from Java concurrency primitives to our concurrency model.

The description of the Java memory model is vague and incomplete [Pug99]. In this work, we assume that updates to values shared between threads take place instantly.

Java is an object-oriented language. Its concurrency primitives rely on runtime system objects for inter-thread signalling. Hence, the situation in real Java programs is somewhat complicated. In particular, any reasonably accurate Java analysis will require alias analysis. We are not aware of any alias analysis work specifically for Java, but there is work on alias analysis in general, as well as pointer analysis for Java [WR99, LR91, EGH94].

6.1.1 The Java concurrency model

The basic Java concurrency primitives are *fork, join*, mutual exclusion (confusingly named synchronization in Java) and wait/notify communication between threads.

In Java, a new thread is spawned by calling the `Thread.start()` method with an object o implementing the `Runnable` interface, or the `start()` method on an object o extending the `Thread` superclass. This causes a new thread to be spawned. It starts its execution in the `run()` method on o.

This model of thread spawning guarantees that intraprocedural analyses need not consider the effects of concurrency on local variables. Any globally-accessible data, of course, can be modified by concurrent threads, necessitating conservative assumptions.

Recall that joining a concurrent thread involves waiting for the concurrent thread to complete its execution before proceeding. In order to join in Java, we call the `join()` method on `Thread` (passing it the `Runnable` object) or on object o extending `Thread`, as appropriate. To do this statically, we must match up the object involved in the `start()` call with the one involved in the `join()` call.

Mutual exclusion in Java is accomplished by obtaining locks. Every Java object has an associated lock. At any point, a thread may request the lock on an object o by putting some statements in a `synchronized(o)` block; once the block finishes, the lock is released. This ensures that the code inside the `synchronized` block is guarded by the lock o; no other thread may obtain that lock. In order to distinguish different locks, a precise alias analysis is required.

Java provides inter-thread signalling with `notify()` and `wait()` functions. Signalling is done through lock objects. Each lock object has an associated wait queue; when a thread waits on an object, it blocks until a `notify()` is called on the same object. Before either waiting for or sending a signal, a thread must possess the corresponding lock.

There are actually two methods which send a signal: `notify()` and `notifyAll()`. The former method arbitrarily picks one thread to continue, while the latter method unblocks all threads waiting on the lock. Even when a thread t is unblocked, it will still need to hold the associated lock to continue: it may no longer be waiting for a `notify`, but another simultaneously-unblocked thread t' might get the lock. In that case, t must wait until t' releases the lock. Usually, notification is used for informing another thread of a change of state; thus, programs should almost always use `notifyAll()` instead of `notify()`, as more than one thread might be waiting on a lock, and all threads should know about the change of state.

Receiving a signal is a multistage process. We divide a `wait()` into three conceptual steps. These steps can be separated into three distinct control-flow graph nodes. The first step, represented by the `wait` node, is to release the lock, so that a notify has a chance to run. Then, the thread sleeps in a `waiting` node, waiting for a `notify` to trigger it. Finally, the thread wakes up in a `notified-entry` node and requests the lock. This allows us to put the $\twoheadrightarrow$ edge from the `notify()` to the `notified-entry` node. Furthermore, we replace the other sequential edge following the `notify()` with a $\twoheadrightarrow$ edge. We also do this for the edge from the `waiting` node to the `notified-entry` node. Note that $\twoheadrightarrow$ can only occur if the `notify` and `waiting` nodes are related by $\bowtie$; otherwise, notification will never occur.

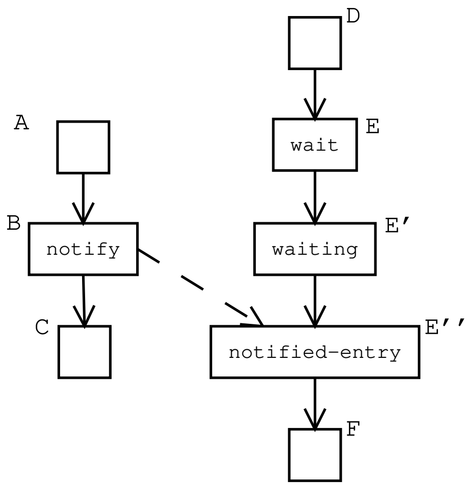


Figure 6.1: A simple Java wait/notify pair

Timeouts In Java, `wait()` can take a parameter indicating how long it is to wait before timing out. Once the specified interval runs out, the thread continues its execution, without notification. We are not able to distinguish between a timeout and a notification. As it is possible that the notification did not occur, we cannot guarantee that $\twoheadrightarrow$ relations hold. In this case, we can neither propagate information, nor can we remove any interleaving operators.

A simple wait/notify pair We discuss the mapping between wait/notify and $\twoheadrightarrow$. Consider the wait/notify pair in Figure 6.1. We assume that we can statically determine that the illustrated notify and wait instructions are matched, and that no other program points act on the same lock.

For our example, it is in fact not true that $B \twoheadrightarrow E'$. Communication is asynchronous, so B might well finish executing before E' has started to wait; in that case, that thread would wait forever once it reaches E'. However, $B \twoheadrightarrow E''$ does hold. When E'' executes, B has certainly already executed. Furthermore, there exists an actual interleaving where B executes, then E''. This matches the semantics of $\twoheadrightarrow$ and we are thus able to assert that $B \twoheadrightarrow E''$.

Note that we need not guarantee that F execute. The $\twoheadrightarrow$ relation states that should F execute, it will follow B.

In this situation, we have the following $\bowtie$ relations. We can assert that $A \bowtie D, A \bowtie E, B \bowtie D, B \bowtie E', C \bowtie D$ and $C \bowtie F$. We draw attention to $C \bowtie D$. The communication is asynchronous, so B and C may both finish before anything in the DEF thread runs.

Multiple locking It may well happen that several different program points are waiting for a signal, or that multiple points are sending signals. This is because any object which has access to the lock object may send a signal on it or receive a signal from it. We now discuss this situation.

If we have several possible program points $w_0, \ldots, w_n$ waiting for a signal, and a single notify point, then we can assert $\twoheadrightarrow$ relations from the single notify point to all of the w's. This will allow us to remove $\bowtie$ relations between successors of the w's and predecessors of the notify. Consider that if any of the w's gets to execute, then it has certainly been preceded by the notify, possibly immediately. Recall that the semantics of Java state that `notify()` arbitrarily chooses one of the waiting threads to be woken up, while `notifyAll()` wakes up all of the threads. Any of the formerly-waiting threads could eventually execute, but certainly not before the notify executes. As long as there is no `notifyAll` statement, we do get to assert that the w_i's are free of $\bowtie$ relations, because only one thread gets woken up.

On the other hand, the presence of multiple program points $n_0, n_1, \ldots, n_k$ sending signals to one, or several, wait statements does not allow us to assert any $\twoheadrightarrow$ relations. Consider two `notify` statements B, H and one wait statement E, as illustrated in Figure 6.2. We cannot tell which `notify` wakes up E. This prevents us from asserting $\twoheadrightarrow$ relations between A and F or between G and F. If B synchronizes with E, then G could be executed immediately after F; conversely, G could executed before E by the scheduler. Hence in this case, $\bowtie$ relations must be preserved between the successors of the waits and predecessors of the notifies.

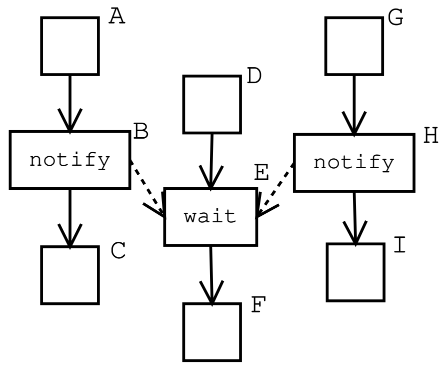


Figure 6.2: Two `notify()` statements act on one `wait()` statement

6.1.2 Computing relations for Java

In work by Naumovich, Avrunin and Clarke [NAC99], an analysis for detecting the may-happen in parallel (MHP) relation for Java programs is presented. This is precisely the $\bowtie$ relation we use in our liftings. Taylor has shown in [Tay83] that computing precisely which statements may happen in parallel is NP-hard in the intraprocedural context; our algorithm computes an approximation. The analysis presented in the work by Naumovich *et al.*, is only applicable to programs where each thread has only one procedure, and where any given thread is only invoked once. It also assumes exact alias analysis for lock and thread objects.

We will first present the MHP algorithm as described in [NAC99], along with an example. We will discuss the effects of imprecision about the identity of lock objects on this algorithm. Finally, we will detail what is needed for the algorithm to deal with the case of programs with multiple procedures per thread, including the case where a procedure is executed by several threads.

This algorithm has been proved to compute a conservative approximation to the ideal MHP information. Its worst-case time complexity is $O(|N|^3)$, where $|N|$ is the number of program statements.

The MHP algorithm

The proposed algorithm looks similar to a standard forward dataflow analysis, but has one important step preventing it from being expressed in the standard dataflow framework. For each statement, the analysis returns a set of statements which may happen in parallel. We also extend it to report on $\twoheadrightarrow$ relations.

Notation A (`obj`, `entry`, `t`) statement denotes a node belonging to thread t requesting the lock for `obj`, while an `exit` routine releases the lock. Earlier, we have alluded to the three-stage `wait` process in Java. This translates to nodes (`obj`, `wait`, `t`), (`obj`, `waiting`, `t`) and (`obj`, `notified-entry`, `t`).

Two kinds of sets are tracked for every node. The M sets represent $\bowtie$ information: if $x \in \mathrm{MHP}(y)$, then $x \bowtie y$. The OUT sets represent information which is to be propagated to successors; facts in $\mathrm{OUT}(x)$ hold immediately after x finishes its execution.

In our liftings, we require both $\bowtie$ and $\twoheadrightarrow$ information. MHP is exactly equivalent to $\bowtie$, while $\twoheadrightarrow$ can be computed by a slight extension of the MHP algorithm.

Symmetrization The $\bowtie$ relation is symmetric, while results of dataflow analyses usually are not. An important step in this algorithm is to impose symmetry: if $n_1 \in M(n_2)$, then we add n_2 to $M(n_1)$. This is crucial to the propagation of information through the graph and prevents it from being expressed in the standard dataflow analysis framework.

MHP Rules The following rules are taken from the original work on the MHP algorithm. First we present the rules for $M(n)$. These sets accumulate from the

OUT sets of the predecessors.

$$M(n) = M(n) \cup \begin{cases} \left(\bigcup_{p \in \text{StartPred}(n)} \text{OUT}(p) \setminus N(\text{thread}(n))\right) & \text{if } n \text{ is } \texttt{begin} \\ \left(\begin{array}{c} \left(\bigcup_{p \in \text{NotifyPred}(n)} \text{OUT}(p)\right) \\ \cap \text{OUT}(\text{WaitingPred}(n)) \end{array} \right) & \\ \quad \cup\ M_{\text{notifyAll}}(n) & \text{if } n \text{ is } \texttt{notified-entry} \\ \bigcup_{p \in \text{LocalPred}(n)} \text{OUT}(p) & \text{otherwise} \end{cases}$$

Every thread begins its execution at a `begin` node. At such nodes, we propagate the OUT information from all of the `start` nodes triggering the execution of the thread, and subtract $N(\text{thread})$, the nodes in the thread being begun. This ensures that nodes that were parallel with the `start` remain parallel with the new thread.

Recall that a `wait` is decomposed into three stages. The `notified-entry` node is the final stage, executing only after some other thread has executed a `notify` on the same lock. At a `notified-entry` node, we add the set of statements concurrent after the predecessors of any matching `notify` (written as notifyPred(n)), as long as these statements were already in the OUT set of the `waiting` node immediately preceding the `notified-entry` node (expressed as WaitingPred(n)).

This clause takes effect only when multiple threads could potentially `notify` to the `wait` in question; otherwise, this intersection is empty, as OUT(WaitingPred(n)) would only contain statements from the notifying thread, while the OUT sets of the notifyPred statements only contain statements from the waiting thread. When the intersection is nonempty, we add to $M(n)$ statements from our predecessor which are also concurrent with the matching `notify` statements. This has the effect of forcing successors of the `wait` to be parallel with the predecessors of the various `notify` statements.

If n is not a `notified-entry` statement, then we set $M_{\text{notifyAll}}(n) = \emptyset$. Otherwise, if n is a `notified-entry` statement on object `obj`, we have:

$$\begin{aligned} M_{\text{notifyAll}}(n) \quad = \quad & \{m \mid m \in (\texttt{obj}, \texttt{notified-entry}, *) \wedge \\ & \quad \text{WaitingPred}(n) \in M(\text{WaitingPred}(m)) \wedge \\ & \quad (\exists r \in N : r \in (\text{obj}, \text{notifyAll}, *)) \wedge \\ & \quad r \in M(\text{WaitingPred}(m)) \cap M(\text{WaitingPred}(n)) \end{aligned}$$

That is, all `notified-entry` statements m on the same lock become concurrent with n, as long as they were already concurrent with n's `waiting` predecessor, and there exists a `notifyAll` statement on lock `obj` which acts on both m and n.

Given the M sets, we can compute the OUT sets for each node. These correspond to statements which are parallel after the execution of n has completed. Each statement has gens and kills:

$$\mathrm{OUT}(n) = (M(n) \cup \mathrm{GEN}(n)) \setminus \mathrm{KILL}(n)$$

The GEN rule is fairly straightforward, handling `start` nodes and `notify` nodes.

$$\mathrm{GEN}(n) = \begin{cases} (*, \mathtt{begin}, \mathtt{t}), & \text{if } n \text{ is the } \mathtt{start} \text{ node for } t \\ \mathrm{NotifySucc}(n), & \text{if } \exists \text{ obj} : n \in \mathrm{notifyNodes}(\mathtt{obj}) \\ \emptyset, & \text{otherwise} \end{cases}$$

The first rule ensures that, after a `start` node, we add the `begin` node for the thread being started. This goes in the GEN rule because we consider that the other thread only starts after the `start` has completed its execution.

If a statement n is a `notify` or `notifyAll` statement on `obj`, we add all of the `notified-entry` successors m, as long as m's `waiting` predecessor is in $M(n)$:

$$\mathrm{NotifySucc}(n) = \{m \mid m \in (\mathtt{obj}, \mathtt{notified\text{-}entry}, *) \wedge \mathrm{WaitingPred}(m) \in M(n)\}$$

This ensures that (only) the successors of a `notify` are concurrent with the successors of a `wait`. The predecessors of the `notify` avoid being concurrent with the successors of the `wait` because the $M(n)$ rule for `notified-entry` prevents any such predecessor from passing through the `notified-entry` node on the `waiting` thread.

In some cases, we can state that after a statement completes, certain other points are no longer concurrent. We present the KILL rule, describing when this can be done.

$$\mathrm{KILL}(n) = \begin{cases} N(t), & \text{if } n \text{ is a } \texttt{join} \text{ on } t \\ \mathrm{Monitor}_{\texttt{obj}}, & \text{if } n \in (\mathrm{obj}, \texttt{entry}, *) \cup \\ & (\texttt{obj}, \texttt{notified-entry}, *) \\ \mathrm{waitingNodes}(\texttt{obj}), & \text{if } (n \in (\texttt{obj}, \texttt{notify}, *) \\ & \qquad \wedge |\mathrm{waitingNodes}(\texttt{obj})| = 1) \\ & \qquad \vee (n \in (\texttt{obj}, \texttt{notifyAll}, *)) \\ \emptyset, & \text{otherwise} \end{cases}$$

If n is a `join` statement, we can safely state that successors of n are no longer concurrent with any nodes from the thread being joined, as `join` guarantees that the joined thread will terminate before execution proceeds.

The KILL rule also handles mutual exclusion. When an `entry` or `notified-entry` statement is executed, then we may assert that no other statements protected by the same lock may execute in parallel. Note that no special treatment is necessary for `exit` nodes; simply not being under mutual exclusion will add the necessary parallel statements to M.

Finally, if n is either a `notify` statement with exactly one successor or a `notifyAll` statement, we kill all matched `waiting` nodes. In either case, we can be certain that the `waiting` statements are going to be woken up; this does not hold for a `notify` with multiple `waiting` successors. Given this information, we know that the `waiting` nodes will complete; they will be followed by the `notified-entry` nodes.

Computing happens-before relations The results of our lifted analyses are tightened when we have $\twoheadrightarrow$ information. Examining the MHP rules allows us to discover places where $\twoheadrightarrow$ can be asserted. At a `join` node, we can assert that the `end` node for the concurrent thread is $\twoheadrightarrow$ related to the `join` node's successor. Also, at a `notified-entry` node n, we have $\twoheadrightarrow$ if n's `waiting` predecessor is concurrent with n, and there is exactly one `notify` or `notifyAll` predecessor. In this situation, we also assert that $\twoheadrightarrow$ holds from the `notify` node to the `notified-entry` node.

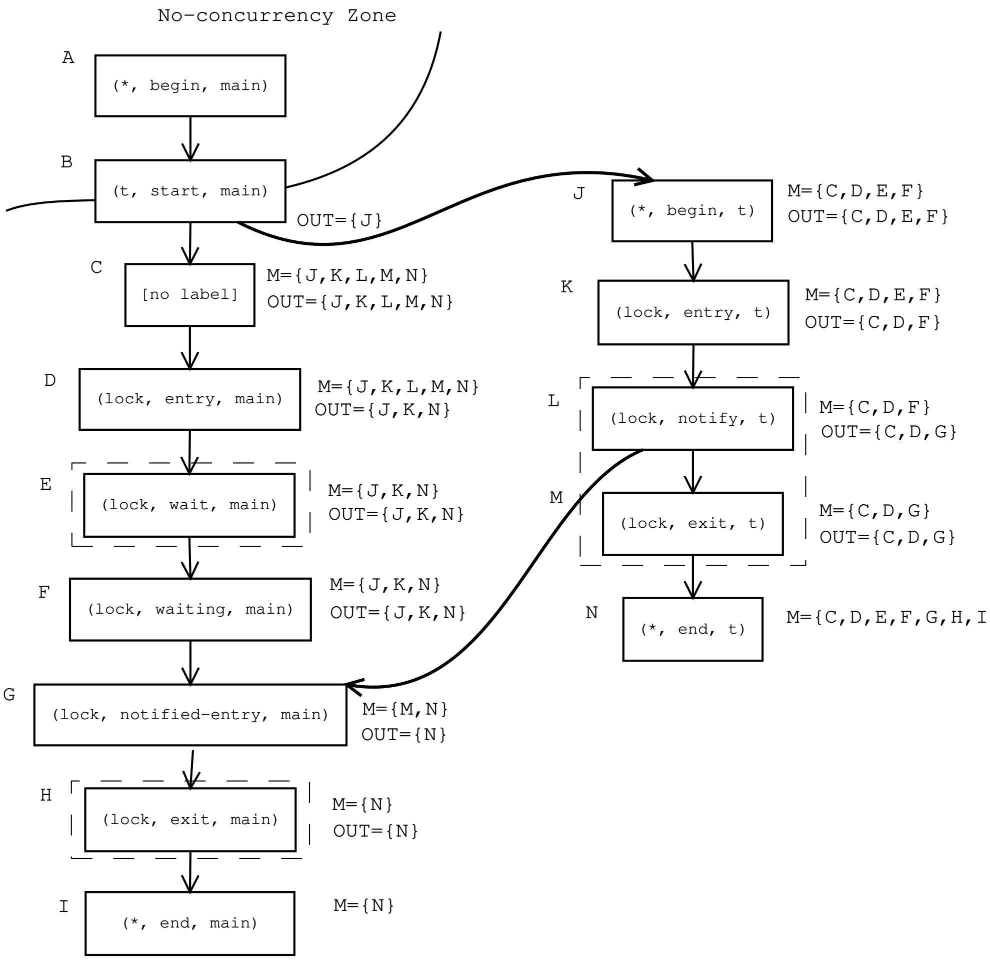


Figure 6.3: MHP analysis on a simple Java wait/notify pair

Example of MHP Computation

Having presented dataflow rules for MHP computation, we illustrate the result of this analysis with a simple example. Figure 6.3 is a simple wait/notify pair, adding some wrapper code to figure 6.1; it also serves to illustrate the tricky propagation of MHP information.

We observe that the `start` node `B` gets `J` in its OUT set, reflecting the fact that the corresponding `begin` node occurs in parallel with the successors of `B`. The `J` propagates to the IN node of `C`, and continues down to the `notified-entry` node. At a `notified-entry` node, parallel nodes must also be parallel with some corresponding `notify` statement, so `J` is stopped there. The symmetrization step ensures that `J` gets `C, D, E,` and `F` in its M set, and all of these are propagated to `K`. At `K`, we exclude `E` because of mutual exclusion. At the `notify` node `L`, we remove the `waiting` node per the rule for M, and add the `notified-entry` node by the GEN rule. This propagates to node `M`. Finally, we hit node `N`, which is parallel with `C`. Since it is not under any monitor, `N` propagates straight through on the `main` thread, from `C` to `I`; symmetrization puts all of the `main` nodes in `N`'s M set.

MHP in the presence of imprecise alias analysis

The MHP algorithm presented above assumes that there is an exact alias analysis, resolving all of the lock and thread objects. It may happen that the available alias analysis is not good enough to resolve all of the lock objects exactly. The original work did not deal with this situation; we discuss what has to be done.

Inexact thread object information In Java, threads are represented as objects. This may result in imprecision about the identity of the thread which gets started. When starting a new thread, we must add to OUT(s) all `begin` nodes for objects possibly aliased to the started thread, as this corresponds to possible flow of control. At a `join` node, on the other hand, we kill all nodes of the joined thread from the OUT set. We can only carry out this kill given exact thread object information; otherwise, there is no guarantee of non-concurrency.

Note that under the assumption that exact alias information is known, it is impossible to express a program with a statically unbounded number of threads: we would have to account for each of the possibly-created threads. We return to this problem in section 6.1.3.

Inexact lock object information Lock objects are used for mutual exclusion and inter-thread communication.

Mutual exclusion is accomplished by requesting the monitor for some object. Inside the protected block, we can assert that no statements under the same monitor may execute in parallel. This can only be asserted for statements which must be under the same monitor; it is not sufficient to have may-aliased information.

For locking, we do require precise alias analysis to assert $\twoheadrightarrow$; otherwise, our deductions are severely limited.

Lock objects are used for the $M(n)$ rule when n is a `notified-entry` node:

$$\begin{aligned} M(n) &= M(n) \cup \left(\left(\bigcup_{p \in \mathrm{NotifyPred}(n)} \mathrm{OUT}(p) \right) \cap \mathrm{OUT}(\mathrm{WaitingPred}(n)) \right) \\ &\quad \cup M_{\mathrm{notifyAll}}(n) \end{aligned}$$

Here, we set NotifyPred(n) to include all statements notifying on an object possibly aliased to the lock object at n. Similarly, at the `notifyAll` rule, we take all nodes possibly invoking `notifyAll` on the same object as the `notified-entry`.

For the GEN(n) rule, recall that we apply the NotifySucc rule for all `notify` and `notifyAll` statements. Let n be a `notify` statement on o. The NotifySucc set must include all objects m such that $m \in (o', \texttt{notified-entry}, *)$ where o is possibly aliased to o', and the `waiting` predecessor of m is parallel to n.

Finally, we consider the KILL(n) rule. For a `join` node, we can only kill $N(t)$ if we are certain about t's identity. Similarly, we can only kill mutually-exclusive statements if we are sure that they are under the same monitor. We also may kill `waiting` nodes only for matching `notify` objects.

6.1.3 Interprocedural MHP analysis

The original work on MHP for Java programs does not discuss threads with multiple procedures; instead, they inline so that there is only one procedure to handle. This is undesirable for analysing real programs; for instance, inlining causes code bloat, and makes subsequent analyses difficult to perform. We will discuss a strategy for analysing programs with procedure calls.

To carry out an interprocedural analysis, we augment the control-flow graph with nodes representing method invocations and returns. Often, making an analysis interprocedural requires some mapping between values in the calling procedure and the called procedure; this is not required for MHP analysis, as no values are propagated, only program points.

Recently, context-sensitive synchronization-sensitive analysis has been shown to be undecidable by Ramalingam [Ram99]; this is caused by the presence of recursion. A context-sensitive analysis is one which only considers paths through the program where method invocations are exactly matched with method returns. Our analysis is not sensitive to calling contexts, thus avoiding this difficulty.

One situation not handled by the original MHP analysis which is especially relevant to interprocedural analysis is the possibility that some statements are executed by two threads. The only way that would happen in the absence of procedures would be a thread that gets spawned multiple times. With procedures, two threads simply need to call the same procedure to produce this behaviour.

No self-parallel statements

We first handle the simpler situation where no statements are self-parallel.

To make our analysis interprocedural, we visit all possibly called procedures (inexact information may arise, for instance, from virtual method invocation) at an method invocation node s, passing the M set from s to the M set for start nodes of all targets. We then collect all OUT sets from return nodes and put them in the OUT set for s. When no statements are self-parallel, we can deduce that no method will be called by multiple threads.

This does not terminate in the presence of recursion. When we visit an invoke site for a method that is already being visited, we store for the recursive method an IN (M) set and an OUT set. Every time this method is possibly called at a statement s, we compare the M set from s with that stored for the start node of the method.

If the stored set contains the one at s, we set the OUT set for s to be the stored OUT set for the method. Otherwise, we propagate the M set from s through the method, merging (with union) the OUT sets from all return statements and storing the merged OUT sets as s's OUT set.

This algorithm is always guaranteed to terminate. For non-recursive programs, the pass through the call graph will be acyclic, so it will make at most one visit to every method; the presence of recursion does not affect termination, because the OUT sets strictly increase, but multiple passes may be required.

Handling self-parallel statements

The MHP algorithm described explicitly assumes that no statement may occur in parallel with any other statement in the same thread. This assumption can be justified when every thread executes exactly one method; in that case, we might consider duplicating the thread bodies for different instances of threads. However, we would like to consider programs where a statically unbounded number of threads can be spawned, in order to deal with programs which use common Java programming idioms.

First, we consider the case where a thread has multiple active instances. For instance, a Java server will often start up a thread to handle each request. Any given instance of a thread may only be started once, but many instances of the thread may be created and started. The MHP analysis described by Naumovich, Avrunin and Clarke, cannot understand such a program: their description of MHP analysis requires that the program being analysed has only at most one instance of any program statement running at any time.

It turns out that, for MHP analysis, we only need to distinguish the case where a thread is started once from the one where a thread is started many times. In the latter situation, we cannot hope to duplicate the thread body once for each possible running thread. We must declare that the nodes from the thread are parallel to themselves.

The rules for M explicitly disallow a thread from executing in parallel with itself, When n is a `begin` node, we have:

$$M(n) = M(n) \cup \left(\bigcup_{p \in \text{StartPred}(n)} \text{OUT}(p) \setminus N(\text{thread}(n)) \right)$$

If we did not remove $N(\text{thread}(n))$, we would allow information about self-parallel statements to flow through the thread. This is what we want.

Within the framework of the original MHP analysis, it was useful to remove nodes in the same thread; after all, in their model, it is an error to start a thread more than once. Hence they can trim their MHP sets by removing relations which are guaranteed not to happen.

The other possibility is a procedure shared between threads. This also leads to nodes being potentially parallel with themselves. However, in that case, the MHP algorithm will simply proceed to add (many) self-parallel nodes. We illustrate this phenomenon in Figure 6.4. Note that the information collected is much degraded in this situation. This is unavoidable unless we collect distinct M sets for each thread which invokes the method. Doing so is similar to conceptually cloning the method for each thread from which the method can be called, although the code need not be duplicated. Collecting distinct M sets gives rise to the situation illustrated in Figure 6.5. Much tighter information can be collected in that case.

6.2 Ada

We discuss Ada concurrency primitives in this section. Like Java, Ada has built-in support for concurrency. There has been some work which corresponds closely to computing our $\rightarrow$ and $\bowtie$ relations for Ada programs; in fact, it predates the work on MHP for Java.

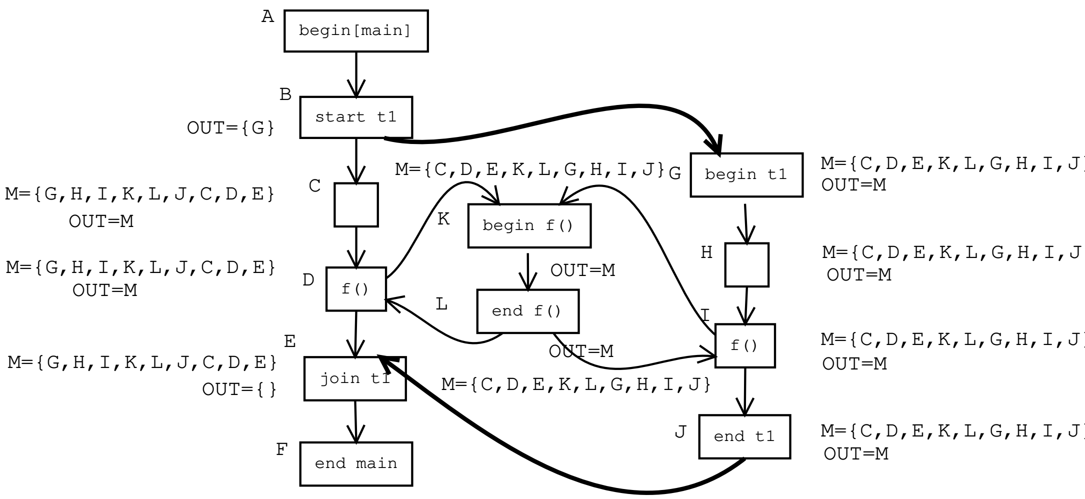


Figure 6.4: MHP analysis on an example where a method is invoked by two distinct methods

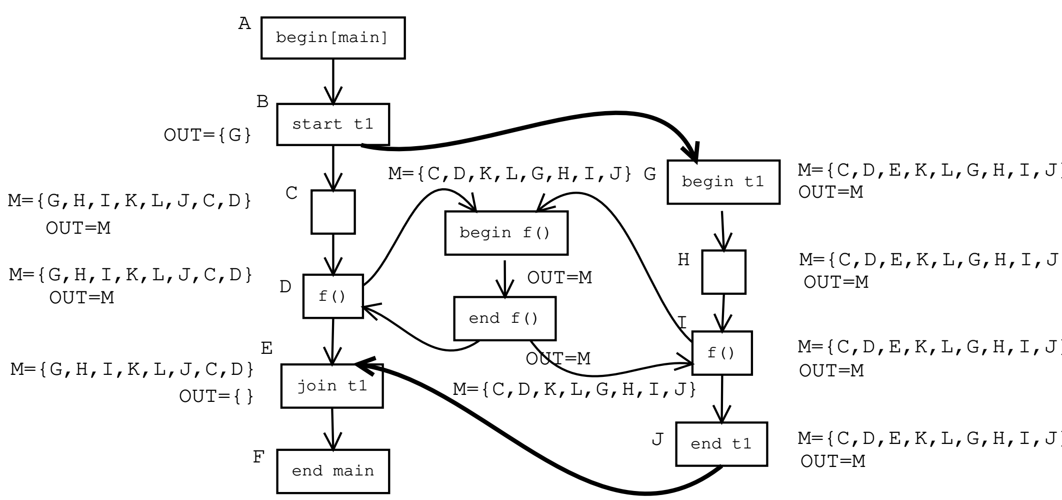


Figure 6.5: MHP analysis on the example of Figure 6.4, distinguishing contexts

6.2.1 The Ada concurrency model

The Ada concurrency model is rather complicated. We will treat a representative subset of the Ada model. Our reference is [BW98], although the Ada Reference Manual [Org95] may also be consulted.

An Ada task is declared with Ada's `task type` construct; instantiations of task types can occur where declarations of other data types can occur. The corresponding tasks are started when the instantiation appears in scope. Tasks can also be created dynamically. For our purposes, we will make the simplifying assumption that all tasks are started simultaneously, when the program starts.

The Ada synchronization primitive is the rendezvous. Ada-style rendezvous is a construct which ensures that in a pair of threads, neither thread proceeds beyond a certain point before the other one does. This communication is synchronous.

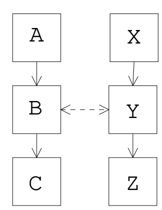


Figure 6.6: Example of Ada Rendezvous

In figure 6.6, we see an example of an Ada rendezvous. There, we know that $A \bowtie X$ and $C \bowtie Z$, but also that $A \to Z, X \to C, B \twoheadrightarrow Z, Y \twoheadrightarrow C, X \twoheadrightarrow B, A \twoheadrightarrow Y$. We omit any relation between B and Y.

Because rendezvous is synchronous, we may remove $\bowtie$ relations from all successors of one node and predecessors of the other when there is communication.

Rendezvous is actually declared in Ada code by using an entry (declared with the task) and an entry call: in the declaration for the task type, the accepted entries are declared:

```
task type Server is
```

```
  entry respond(ID: in Integer);
end Server
```

and any client can call the entry:

```
  Server.respond(3);
```

There is also a *select* mechanism, which accepts a number of entry types and chooses one of them; we assume that it acts nondeterministically.

6.2.2 Finding relations in Ada programs

The Ada model is also amenable to finding $\bowtie$ and $\twoheadrightarrow$ relations. We now present an algorithm from [NA98] for doing so. Like the Java analysis for Must Happen in Parallel, it is only applicable to programs with one procedure per thread and no self-parallel units. It is much simpler, because the Ada rendezvous construct is much simpler than Java's concurrency support.

The analysis assumes that all threads are represented by separate control-flow graphs, and that all of the graphs are available statically. There are two types of nodes in each CFG: local nodes, corresponding to usual Ada statements, and communication nodes, corresponding to rendezvous. A communication node has two predecessors, one for each participant in the rendezvous, and two successors. There is also an *initial* node for the whole program; it is the predecessor for all of the task begin nodes.

In this analysis, we track IN and M sets for each node. The IN set collects information from the predecessors of a node, while the M set tracks the current estimate of nodes which may happen in parallel. An additional bit of information we track is a *Reach* bit for every communication node. This bit has an initial value of false, and is set to true if both rendezvous predecessors are parallel to each other. Clearly, this is a necessary condition for the rendezvous to proceed.

The rule for the IN set at a local node merges the information from all predecessors:

$$\text{IN}(n) = \bigcup_{p \in \text{Pred}(n)} M(p)$$

while at a communication node, we require both predecessors to possibly execute:

$$\text{IN}(n) = \begin{cases} \bigcap_{p \in \text{Pred}(n)} M(p) & \text{if } \mathit{Reach}(n) \\ \emptyset & \text{otherwise} \end{cases}$$

To obtain the M set from the IN set, we calculate a GEN set, so that

$$M(n) = \text{IN}(n) \cup \text{GEN}(n)$$

Defining the GEN set first requires us to set up an P set for each statement n. P contains the initial node, as long as there is a path from the initial node to n in some CFG; it also contains all communication nodes with a path to n which currently have their *Reach* bit set to true. Intuitively, P represents the set of currently-reachable nodes. We set

$$\text{GEN}(n) = \left(\bigcup_{p \in P} \text{Succ}(p) \right) \setminus \{ m \mid m \text{ is in the same task as } n \}$$

Once again, we assume that no statements are self-parallel. We generate all (transitive) successors of the statements in P, as long as they are not in the same thread as n.

The final step in this algorithm is to symmetrize. If $n_1 \in M(n_2)$, then we ensure that $n_2 \in M(n_1)$. Again, this necessary step prevents the algorithm from being stated as a standard dataflow analysis.

This analysis can be implemented so that it terminates in $O(|N|^3)$ time (where $|N|$ is the number of nodes in the program) and is guaranteed to compute a conservative approximation of the perfect MHP information.

We have already discussed how to extend the Java MHP analysis to handle programs with multiple procedures and self-parallel units. The situation for Ada programs is not very different; we omit discussion of these issues.

6.3 CML

Another language with concurrency primitives is CML. As it is a higher-order language, its analysis is somewhat more complicated than that for languages like Java and Ada; however, here we are solely discussing its concurrency model.

CML uses synchronous communication. Hence, an analysis similar to the one we have presented for Ada in section 6.2 would be applicable, once the higher-order features of CML are tamed.

6.3.1 The CML concurrency model

We present the CML concurrency primitives in figure 6.7. A discussion of CML's higher-order concurrency can be found in [Rep92].

Figure 6.7: CML concurrency operations

```
type thread_id
type 'a chan
type 'a event

val spawn : (unit -> unit) -> thread_id

val channel : unit -> 'a chan
val recv    : 'a chan -> 'a
val send    : ('a chan * 'a) -> unit

val recvEvt : 'a chan -> 'a event
val sendEvt : ('a chan * 'a) -> unit event

val guard   : (unit -> 'a event) -> 'a event
val wrap    : ('a event * ('a -> 'b)) -> 'b event
val choose  : 'a event list -> 'a event

val sync    : 'a event -> 'a
val select  : 'a event list -> 'a
```

CML has four standard concurrency primitives: spawn, channel, recv and send.

The spawn primitive creates a new thread and starts execution of a given function in that thread. The channel function creates a new channel and returns it. These channels can be used to send and receive messages.

In CML, send and receive are synchronous operations; of course, receive blocks until a message is sent, but send also blocks until that message is received. If we know that a send and receive are on the same channel, and that the receive is the only one on that channel, then we can assert a $\twoheadrightarrow$ relation between these statements. When we do so, we also assert $\twoheadrightarrow$ to the successor of the `send` and `receive` nodes.

The other CML primitives involve *events*. CML allows the construction of receive and send events. These events are data values that can be acted on later by the sync primitive: synchronizing on a receive event causes a receive to take place. Furthermore, the guard function creates an event out of a function promising to return an event. The wrap function takes an event e and a function f and returns another event. When the returned event is synchronized on, the event e gets synchronized on, and then f is applied to the result of e. The final event operation is choose. It takes a list of events and, upon synchronization, nondeterministically chooses one event from the list to be synchronized on.

To act on events, we have the sync function, which takes an event and carries out the associated action. There is also the select function, which is equivalent to composing sync and choose.

6.3.2 Detecting causality in CML programs

The highly dynamic nature of ML programs makes their analysis quite difficult. Nevertheless, there has been work on optimizing ML programs [TMC+96]. We will discuss what a compiler must know to be able to assert $\twoheadrightarrow$ and $\bowtie$ relations.

For our causality analysis, it is imperative to resolve the communication channels as accurately as possible. Such disambiguation is required in order to impose an ordering on statements in different threads: there is no ordering if we cannot determine that two program points are acting on the same channel.

A synchronous communication model imposes a symmetry between the receive and send actions. A send and a receive on the same channel are equivalent in terms of temporal ordering. However, in the presence of multiple send events or multiple receive events, we notice that sends and receives are queued separately; a receive event does not enable another receive event.

It is fairly simple to deal with the usual concurrency primitives `spawn`, `recv` and `send`, as long as we can reliably differentiate the various channels used for communication. After a `spawn` statement, the evaluation of its argument proceeds in parallel, so we add $\bowtie$ relations between the successors of the `spawn` and the nodes of f. If a `recv` statement r and a `send` statement s are associated with the same channel, and we can guarantee that these are the only statements acting on this channel, then we conceptually split r into two parts, r_0 and r_1, and s into s_0 and s_1. We can establish that $r_0 \twoheadrightarrow s_1$ and $s_0 \twoheadrightarrow r_1$. In this situation, no $\bowtie$ relations hold between the predecessors of r_0 and the successors of s_1, nor between the predecessors of s_0 and the successors of r_1.

There may be multiple sends or multiple receives on one channel. If that may happen, we no longer have any causality information, because any send may unblock a receive, and any receive may block a send.

A send or receive with ambiguous channel information must be treated as if it could act on any of the possible channels. If we can assert that we have only one send and one receive action on all involved channels, then we can still assert $\twoheadrightarrow$.

When dealing with events, we require even more information at compile-time. In order to count the number of send events versus the number of receive events, we must statically identify which types of events can reach a `sync` instruction. We still require that the channel to which the `sync` refers be known at compile time. Given this information, we deal with events in the same way as we dealt with the primitives.

In this section, we have presented principles for detecting temporal relations in

CML programs. An algorithm for computing happens-before information in CML programs requires completely different tools than those we have presented in the rest of the work; it is outside the scope of this thesis.

Chapter 7

Related Work

There have been previous investigations of several of the topics discussed in this thesis. The determination of causality relations has been studied by many other researchers. Flow analysis has a sound theoretical background; we provide some references to the classical works. Various analyses for concurrent programs have been proposed in the past. We will examine what has been done in the past.

7.1 Ordering of program events

The concepts of *independent* and *proximal* statements were discussed in Chapters 2, 3, and 6. The precise computation of this information was shown to be NP-hard by Taylor [Tay83]. An early treatment of these issues occurs in work by Callahan and Subhlok [CS88]. It discusses parallel FORTRAN programs and proposes an analysis for estimating when one block of FORTRAN statements must precede another block. In their work, blocks in this situation are called “Preserved”; we can see that this is analogous to the happens-before relation $\rightarrow$. Their analogue to proximal instructions is the notion of “co-executable” blocks. Later, Duesterwald and Soffa proposed an algorithm to order the statements of concurrent Ada programs with procedures [DS91]. There has been work by Ryder and Masticola [MR93] on non-concurrency analysis of Ada programs; this work identifies statements which are not independent. The intraprocedural approximations for $\bowtie$ in Java and Ada programs presented in this work are due to Naumovich, Avrunin and Clarke [NA98, NAC99]; they are currently the state of the art, even if they do not deal with procedures or uncertainty in the synchronization information.

Recently, Ramalingam has shown that interprocedural context-sensitive synchronization-sensitive analysis is undecidable [Ram99]. Our interprocedural analysis to detect the ordering of program events is not context-sensitive.

7.1.1 Alias analysis

Our may-happen in parallel analysis for Java relies on adequate alias analysis. We are not aware of sophisticated alias analyses for Java; however, a type-based approach to alias analysis has been implemented in the Jalapeño dynamic optimizing Java compiler for use in dependence analysis. It is described in [CPS+99]. There are also descriptions of alias analysis in other contexts [LR91, EGH94].

7.2 Foundations of flow analysis

Dataflow analysis is by now a standard technique used in optimizing compilers to detect properties of programs. In section 2.3 we described standard dataflow analysis for sequential programs. A classical work by Kildall [Kil73] shows that if all flow functions are distributive (in our terminology, multiplicative or additive), then the standard iterative algorithm computes the maximum fixed-point solution (MFP) and that this is equal to the meet-over-all-paths (MOP) solution. Furthermore, Kam and Ullman showed in [KU77] that as long as the flow functions are monotone, the standard algorithm at least computes the MFP solution.

The authoritative work mapping the results of flow analysis to information about actual executions is the paper on abstract interpretation of Cousot and Cousot [CC77]. The idea is that the actual computations take place in some universe; we reason about some more limited universe, and abstract interpretation provides the bridge between these two universes. In particular, abstract interpretation allows us to show that the results in the more limited universe are still consistent with the semantics of a language.

7.3 Analysis of concurrent programs

We are aware of some analyses specifically for multithreaded programs. Some analyses consider properties specific to concurrent programs; static analyses for deadlock detection, for instance, have no sequential analogue.

The earliest dataflow analysis of parallel programs of which we are aware was by Grunwald and Srinivasan [GS93]; this work studied parallel FORTRAN programs and solving the reaching definitions problem in the context of post/wait synchronization, which is similar to wait/notify synchronization. However, there were several limitations in that work. For instance, it assumed that threads were data independent; changes in shared state do not propagate to concurrent threads until synchronization takes place. There has also been work on developing a Concurrent Static Single Assignment form by Lee, Midkill and Padua [LMP97]; they apply this CSSA form to constant propagation. Novillo, Unrau and Schaeffer [NUS98] have extended CSSA to handle mutual exclusion. More recently, there has been work by Rinard and Rugina [RR99] on pointer analysis for multithreaded programs.

Perhaps the work closest to ours is by Knoop, Steffen and Vollmer [KSV96]. They have proposed a lifting for sequential analyses. As in our work, they propose a mechanism to automatically produce the corresponding analysis for a concurrent program, given the sequential analysis; it proves that the lifting is efficient and optimal. Their notion of optimality corresponds to our notion of tightness. This lifting has been applied to partial redundancy elimination [KS99]. However, there are several limitations to their work. It is only applicable to bitvector analyses; we provide sound approximations for non-bitvector analyses, like generalized constant propagation. Their lifting only applies to intraprocedural analyses. Furthermore, the model of concurrency studied in their work is quite simple; they only consider forks and joins. Our work deals with concurrency primitives which are easily mapped to those in real languages.

There has also been work on interprocedural analysis of concurrent programs by Seidl and Steffen [SS00]. Their work also deals with some non-bitvector analyses: they simply require that the abstraction domain have finite height, that it be distributive, and that flow functions have the form $f(x) = (a \sqcap x) \sqcup b$. However, they only treat the fork-join model of concurrency; ignoring synchronization leads to a loss of precision.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis, we have provided a general framework that overcomes the limitations to optimizing concurrent programs that are inherent in traditional compiler technologies. We have constructed this framework so that our techniques can be applied without requiring a large-scale retooling of current compiler technology; our work makes standard compiler analyses for sequential programs applicable to concurrent programs. More precisely, we have provided a general lifting permitting the application of standard sequential dataflow analyses to the concurrent setting, taking into account the synchronization structure of the program under consideration.

The main steps in accomplishing this goal and demonstrating its practical applications were as follows.

- First, we presented background material about causal ordering and sequential flow analysis in Chapter 2. We introduced notions central to this work, namely the happens-before relation and the concept of independent instructions. Sequential traces were used to revisit some of the classical results on the flow analysis of sequential programs – we showed the soundness and tightness of the usual flow analysis techniques – as a prelude to our treatment of concurrent programs.

- We next discussed how concurrent traces are obtained from the causal relations between statements in Chapter 3. We introduced the additional causal relation $\bowtie$ which holds between proximal statements. We described how the causal relations determine the set of executions. Our relations improve on previous work, where the effect of synchronization was ignored; in our approach, fewer statements are unnecessarily declared to be proximal. Finally, we discussed a set of concurrency features in programming languages and described how they can be expressed using our concurrency relations.

- We then proceeded in Chapter 4 to the main result of this work. We presented general techniques making it possible to reuse sequential flow analyses in a concurrent setting. First, we proposed a naive lifting, and proved that under certain conditions, the naive lifting is tight. However, the naive lifting is not always tight. For bitvector analyses, we proposed a more sophisticated lifting, and showed that it was tight.

- Our lifting is easy to apply. We demonstrated this in Chapter 5 by lifting several common flow analyses: namely, available expressions, live variables and generalized constant propagation. The lifted analyses were used on several examples of concurrent programs, containing nested thread spawning and interthread synchronization.

- Finally, we addressed the problem of detecting causal relations in real programs written in Java and Ada, and presented the CML concurrency primitives, in Chapter 6. For Java and Ada, we recapitulated past work which identified statements which May Happen in Parallel. We extended the MHP analysis for Java to make it more applicable to real programs. In particular, we described an approach for handling programs with procedures. We also analyse the situation where concurrency relations are not exactly known.

8.2 Future work

Having provided a framework for the analysis of concurrent programs, many interesting problems are raised.

Implementation We have proposed a number of algorithms in this thesis. They should be implemented.

In particular, the intraprocedural May Happen in Parallel analysis has been implemented by Naumovich, Avrunin and Clarke. We proposed an extension to this analysis, allowing it to handle programs with procedures. The practical behaviour of this interprocedural analysis should certainly be examined, and the results compared with ideal results; those can be obtained with an exponential-time algorithm, if the program is sufficiently small.

Once the MHP algorithm is implemented, we are ready to implement lifted analyses and optimizations based on them. The Soot framework is suitable for performing such experiments on Java code, after a suitable benchmark set is assembled.

Interprocedural analysis In this work, we have presented a lifting for intraprocedural flow analyses. To get more precise information, we must analyse the effects of procedures. We are currently extending our lifting to deal with various types of interprocedural analyses.

Bibliography

[BW98] Alan Burns and Andy Wellings. *Concurrency in Ada*. Cambridge University Press, second edition, 1998.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

[CPS+99] Craig Chambers, Igor Pechtchanski, Vivek Sarkar, Mauricio Serrano, and Harini Srinivasan. Dependence analysis for java. In *1999 Workshop on Languages and Compilers for Parallel Computing*, 1999.

[CS88] David Callahan and Jaspal Subhlok. Static analysis of low-level synchronization. In *Proceedings of the SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*, pages 100–111, 1988.

[DS91] Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data flow framework. In *Proceedings of the ACM SIGSOFT Fourth Workshop on Software Testing, Analysis and Verification*, pages 36–48, Oct 1991.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers.

In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 242–256, Jun 1994.

[GJS00] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification, Second Edition*. Addision-Wesley Longman, 2000.

[GS93] Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In PPOPP93 [PPO93], pages 159–168.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conf. Record of the ACM SIGACT/SIGPLAN Symp. on Principles of Programming Languages*, pages 194–206, Boston, MA, Oct 1973.

[KS99] Jens Knoop and Bernhard Steffen. Code motion for explicitly parallel programs. In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99), Atlanta, Georgia, USA, May 4-6*, volume 34 of *ACM SIGPLAN Notices*, pages 13–24. ACM, August 1999.

[KSV96] Jens Knoop, Bernhard Steffen, and Jürgen Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *Transactions on Programming Langauges and Systems*, 18(3):268–299, May 1996.

[KU77] John B. Kam and Jeffrey D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:306–317, 1977.

[Lam77] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the A.C.M.*, 21(7):558–565, 1977.

[Lam00] Patrick Lam. Common subexpression removal in Soot. Technical Report 2000-1, McGill University, Sable Research Group, May 2000. Available at `http://www.sable.mcgill.ca/publications`.

[LMP97] Jaijin Lee, Sam Midkiff, and David A. Padua. Concurrent static single assignment form and constant propagation for explicitly parallel programs.

In *Proc 10th Workshop on Languages and Compilers for Parallel Computing*, Aug 1997.

[LR91] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem taxonomy. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 93–103, Jan 1991.

[MR93] Stephen P. Masticola and Barbara G. Ryder. Non-concurrency analysis. In PPOPP93 [PPO93], pages 129–138.

[NA98] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the 6th International Symposium on Foundations of Software Engineering*, pages 24–34, Nov 1998.

[NAC99] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing *MHP* information for concurrent Java programs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 399–410, May 1999.

[NUS98] Diego Novillo, Ron Unrau, and Jonathan Schaeffer. Concurrent SSA form in the presence of mutual exclusion. In *Proceedings of the 1998 International Conference on Parallel Processing*, pages 356–364, 1998.

[Org95] International Standards Organisation, editor. *Ada Reference Manual: Language and Standard Libraries.* International Organisation for Standardisation and International Electrotechnical Commission, 1995. International Standard ISO/IEC 8652:1995.

[PPO93] *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, May 1993.

[Pug99] William Pugh. Fixing the Java memory model. In *Proceedings of the ACM 1999 Conference on Java Grande*, pages 89–98, Jun 1999.

[Ram99] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. Technical Report RC21493 (96967), IBM T.J. Watson Research Centre, May 1999.

[Rep92] J. H. Reppy. *Higher-order Concurrency.* PhD thesis, Cornell University, June 1992. Available as Cornell CS Technical Report 92-1285.

[RR99] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, pages 77–90, May 1999.

[SS00] Helmut Seidl and Bernhard Steffen. Constraint-based inter-procedural analysis of parallel programs. In *9th European Symposium on Programming (ESOP)*, volume 1782 of *Lecture Notes in Computer Science*, pages 351–365, March 2000.

[Tay83] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[TMC+96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[VCH96] Clark Verbrugge, Phong Co, and Laurie Hendren. Generalized constant propagation: A study in C. In *Proceedings of the 1996 International Conference on Compiler Construction*, number 1060 in Lecture Notes in Computer Science, pages 74–90. Springer, Apr 1996.

[VR00] Raja Vallée-Rai. Soot: A Java bytecode optimization framework. Master's thesis, McGill University, July 2000.

[Wal84] Robert M. Wald. *General Relativity.* The University of Chicago Press, 1984.

[WR99] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, November 1999.