Precise Interprocedural Dependence Analysis of Parallel Programs [1]

Markus Müller-Olm [2]

March 27, 2002

[1]This report is a pre-print of material from the author's habilitation thesis.

[2]The work was supported by the RTD project IST-1999-20527 "DAEDALUS" of the European FP5 programme.

Contents

Abstract

Interprocedural detection of copy constants and elimination of faint code in parallel programs are undecidable problems, if basic statements are assumed to execute atomically [26]. In this paper we show that these problems become decidable, if this assumption is abandoned. So, the (unrealistic) idealization from program verification "atomic execution of basic statements" introduced in order to simplify matters, actually increases the difficulty of these problems from the point of view of program analysis: amazingly these problems become more tractable if we adopt a less idealized, more realistic model of execution.

We propose an effective abstract domain of *sets of dependence traces* that allows to perform a precise interprocedural dependence analysis in (non-atomically executing) parallel programs. The main idea is to trace sequences of dependences mediated successively by program executions. We define operations on sets of dependence traces and show that they precisely abstract the corresponding operations on sets of non-atomic runs. Most importantly, we define a precise abstract interleaving operator. Relying on these operations, we can analyze dependences by means of an abstract interpretation of constraint systems that characterize certain sets of program executions. The result of the dependence analysis can in turn be used to detect all copy constants and to eliminate faint code.

The presented algorithms have exponential worst-case run-time. Indeed, we show that both detection of copy constants and elimination of faint code remain intractable when the atomic execution idealization is abandoned already for parallel programs without loops or procedures. Nevertheless, we believe that refinements of the technique underlying dependence traces can lead to practically interesting algorithms that are much more precise than existing algorithms in which interference is treated rather pessimistically. While the run-time of the algorithms is exponential in the number of program variables, it is *polynomial in the program size*. Hence, they are polynomial-time algorithms if the number of program variables is bounded. An interesting direction for future research is to devise algorithms that use the expensive dependence trace technique only for treating propagation via global variables and combine this with a cheap sequential technique for propagation via (thread- or procedure-) local variables. This seems promising because most variables are local in practice.

Chapter 1

Introduction

1.1 Motivation

Exact Analysis of Parallel Programs

Automatic analysis of parallel programs is known as a notoriously hard problem. A well-known obstacle is the so-called *state-explosion problem*: the number of (control) states of a parallel program grows exponentially with the number of parallel components. Therefore, most practical flow analysis algorithms of concurrent programs conservatively approximate the effects arising from interference of different threads in order to achieve efficiency. An excellent survey on practical research towards analysis of concurrent programs with many references is provided by Rinard [31]. In contrast to this research, we are interested in analyses of parallel programs that are *exact (or precise)* except of the common abstraction of guarded branching to non-deterministic branching that is well-known from analysis of sequential programs.

Surprisingly, certain basic but important dataflow analysis problems can still be solved precisely and efficiently for programs with a fork/join kind of parallelism. Results of this kind have been achieved by generalizing the fixpoint computation techniques common in classic dataflow analysis of sequential programs [21, 20, 33] and by automata-theoretic techniques [9, 10]. The most far-reaching result has been achieved by Seidl and Steffen [33]. They show that all *gen/kill problems*[1] can be solved interprocedurally in fork/join parallel programs efficiently and precisely. This comprises the important class of *bit-vector analyses*, e.g., live/dead variable analysis, available expression analysis, and reaching definition analysis [24]

[1]Gen/kill problems are characterized by the fact that all transfer functions are of the form $(\lambda x : (x \wedge a) \vee b)$, where a, b are constants from the underlying lattice of dataflow facts.

Complexity- and Recursion-Theoretic Limits

In view of these results it is interesting to ask whether there are richer classes of dataflow problems that can precisely be solved for parallel programs. For answering this question we investigated the complexity of various data-flow analysis questions that may be seen as canonic representatives of the next level of difficulty beyond gen/kill problems [25, 26]. Specifically, we looked at detection of copy constants [11] and truly live variables and at program slicing [38, 35]. Detection of truly live variables is the crucial analysis problem underlying faint code elimination [12]. In the sequential setting these problems give rise to *distributive* dataflow frameworks on lattices with small chain height and can thus—by the classic result of Kildall [19, 24]—completely and efficiently be solved by a fixpoint computation.

In [26] we show by means of a reduction from the halting problem for two-counter machines that all the problems mentioned in the previous paragraph are undecidable in parallel programs with procedures (parallel interprocedural analysis). Moreover, these problems are PSPACE-complete in case that there are no procedure calls (parallel intraprocedural analysis), and still (co-)NP-complete if also loops are abandoned (parallel acyclic analysis). The latter results are shown by means of reductions from the intersection problem for regular and star-free regular expressions, respectively. These results render the possibility of complete and efficient dataflow algorithms for parallel programs for more extensive classes of analyses unlikely even for loop-free programs, as it is generally believed that the inclusions P $\subseteq$ (co-)NP $\subseteq$ PSPACE are proper.

The prototypic framework in which these results are obtained poses only rather weak requirement such that the results apply to many concurrent programming languages. In particular the results are independent of synchronization operations in contrast to previous intractability and undecidability results for flow analysis in parallel languages [34, 30]. They may also be compared to undecidability of LTL model-checking for parallel languages (even without synchronization) as proved by Bouajjani and Habermehl [4]. While they also consider a parallel language without explicit synchronization operations, they use the LTL formula to synchronize the runs of two parallel threads that simulate a two-counter machine.

In order to perform our reductions without relying on synchronization we use a subtle technique involving re-initialization of variables. In all reductions programs are constructed in such a way that certain *well-behaved runs* simulate some intended behavior, e.g., the execution sequences of the given two-counter machine in the undecidability proof. But we cannot avoid that the constructed programs have also certain runs that bear no correspondence to the behavior to be simulated, we may call them *spurious runs*. One would use synchronization to exclude spurious runs but in the absence of synchronization primitives this is not possible. In order to solve this problem, we ensure by well-directed re-initialization of variables that the spurious runs do not contribute to the answer of the analysis problem in question. Intuitively, one may interpret this well-directed re-initialization of variables as a kind of "internal synchronization". However, other than synchronization well-directed re-initialization does not prohibit the occurrence of spurious runs; it only ensures that they do not influence the analysis result.

Contribution of this Report

In the reductions mentioned above we assume that assignments execute atomically. This idealized assumption is quite common in the literature on program verification as well as in the theoretical literature on flow analysis of parallel programs. However, in a multi-processor environment where a number of concurrently executing processors accesses a shared memory, it is often unrealistic because assignments are broken into smaller instructions prior to execution. This is explained in more detail in Chapter 3.

Surprisingly, the reductions of [26] break down when the atomic execution assumption for assignment statements is abandoned. Without assuming atomic execution of assignments the subtle game of re-initialization of variables that is crucial for putting the reductions to work can no longer be played, which is illustrated by an example program in Section 3.2. In itself this does not imply that the hardness results are no longer valid: there could be reductions employing other techniques.

In this paper we show that interprocedural detection of copy constants and faint code elimination becomes indeed decidable if the atomic execution assumption is abandoned. Recall that these problems are undecidable if this assumption is adopted. Specifically, we develop EXPTIME-algorithms for these problems. So, the (unrealistic) idealization from program verification "atomic execution of assignment statements" that presumably simplifies matters actually increases the difficulty of these problems from the program analysis point of view: amazingly these problems become more tractable if we adopt a less idealized, more realistic view of execution.

We propose an effective abstract domain of *sets of dependence traces* that allows to perform a precise interprocedural dependence analysis in (non-atomically executing) parallel programs. The main idea is to trace sequences of dependences mediated successively by program executions. We define operation on sets of dependence traces and show that they precisely abstract the corresponding operations on sets of non-atomic runs. Most importantly, we define a precise abstract interleaving operator. Relying on these operations, we can analyze dependences by means of an abstract interpretation of constraint systems that characterize certain sets of program executions. The result of the dependence analysis can in turn be used to detect all copy constants and all truly live variables. The latter can be exploited for faint code elimination.

The resulting algorithms have exponential worst-case run-time. Indeed, we show that detection of copy constants and faint code elimination remain intractable problems even when the atomic execution idealization is abandoned. This holds already for parallel programs without loops or procedures. Nevertheless, we believe that refinements of the technique underlying dependence traces can lead to practically interesting algorithms that are much more precise than existing algorithms in which interference it treated rather pessimistically. While the run-time of the algorithms is exponential in the number of program variables, it is *polynomial in the program size*. Hence, they are polynomial-time algorithms if the number of program variables is bounded. An interesting direction for future research is to devise algorithms that use the expensive dependence trace technique only for tracing propagation via global variables and combine this with a cheap sequential technique for propagation via (thread- or procedure-) local variables. This seems promising because most variables are local in practice.

Throughout this report we assume that the reader is familiar with the basic techniques and results from the theory of computational complexity [28, 18], program analysis [27, 1, 15, 24], and abstract interpretation [7, 8]. In the remainder of this chapter we provide a high-level overview on this report in order to allow the reader an easier access.

1.2 Overview

Constraint-Based Analysis of Parallel Programs

The core of the algorithms to be developed follows the constraint-based approach to program analysis. Constraint-based program analysis provides a framework to develop analyses and argue about their correctness and completeness. The main idea is to perform the analysis by means of a (precise) abstract interpretation of a constraint system characterizing certain sets of runs (=program executions). To apply it to the analysis of *parallel programs* has been pioneered by Seidl and Steffen [33]. In order to come to grips with parallel composition, new operators on run sets are used that are not needed in systems for sequential programs. Seidl and Steffen [33] give a constraint system for *reaching runs* in parallel programs (based on a constraint system for so-called *same-level runs*). The new operators are an interleaving operator $\otimes$ and prefix and postfix operators *pre* and *post*. We work with the same signature here. In general, it is not possible to give precise interpretations of these new operations for arbitrary dataflow frameworks. But Seidl and Steffen show that for gen/kill dataflow problems this can be done. Note that the frameworks for copy constant and truly live variable detection do *not* belong to this class.

In Chapter 2 we define parallel flow graphs as our model of concurrent programs. We furnish them with an operational semantics, and define constraint systems characterizing various sets of runs: same-level and inverse-same-level runs, reaching and terminating runs, and bridging runs. For the moment, we still assume atomic execution of base statements. Same-level and reaching runs are already found in Seidl and Steffen's exposition, and they also indicate that inverse-same-level and terminating runs can be obtained by duality. Bridging runs, however, are new. A further innovation is that, in contrast to Seidl and Steffen, we relate the constraint systems to the underlying operational semantics instead of postulating them. In our opinion this clarifies what exactly is specified by the constraint systems and also helped to uncover and correct a subtle error in their treatment of non-reachable program points. While a detailed understanding of the other sets of runs is not necessary at the moment, we must explain bridging runs.

A bridging run from a program point u to another program point v is a sequence of flow graph edges (or atomic actions) that can bring us from a configuration in which control is at program point u to a configuration in which control is at program point v. Why are we interested in bridging runs? We call a pair of program variables (x, y) a dependence and say that a given run mediates the dependence (x, y) if the value of y after the run depends on the value of x before the run where we judge dependences syntactically (statically) rather than semantically. If we can determine the dependences mediated by bridging runs we can indirectly answer certain program analysis questions. In particular this information suffices to detect copy constants and truly live variables.

Non-atomic Execution

In Chapter 3 we explain, why atomic execution is not a realistic assumption on program execution and motivate and define a non-standard interpretation for the operators and constants used in the constraint systems for parallel programs. This non-standard interpretation captures non-atomic execution of base statements. The idea is to break base statements into atomic actions of smaller granularity and to use an interleaving semantics on these atomic actions. By interpreting the constraint systems from Chapter 2 with the new interpretation, we get run sets that capture non-atomically executing base statements. These run sets are taken as the reference semantics for analysis of parallel programs when the atomic execution assumption is abandoned.

Dependence Traces

Unfortunately, we cannot obtain the dependences of the interleaving $R_1 \otimes R_2$ of two (non-atomic) run sets from the dependences of the two run sets R_1 and R_2: we can invent run sets that have the same dependences but behave differently when interleaved with other run sets. Therefore, we need a more informative abstract domain that allows to record more information than just dependences.

This domain is the topic of Chapter 4. Let us give a rough description of the ideas underlying this domain. A more extensive explanation and motivation is provided by the introduction to Chapter 4 and the body of that chapter gives a full technical account with many examples.

The basic idea is to collect not just dependences but *dependence sequences*. These are sequences of dependences that can successively be mediated by a run. For example, the run $r_1 = \langle c := b, e := d \rangle$ has $\langle (b, c), (d, e) \rangle$ as one of its dependence sequences. This dependence sequence plays a dual role: it captures, on the one hand, the potential of r_1 to mediate the dependence (b, e) if it runs in parallel with a run that can fill the 'gap' between c and d (like, e.g., $r_2 = \langle d := c \rangle$) and, on the other hand, its potential to successively fill the 'gaps' (b, c) and (d, e) in a run running in parallel (like, e.g., in $r_3 = \langle b := a, d := c, f := e \rangle$). This idea needs to be refined further: we must also collect certain information about initial and final transparency of runs, which results in the notion of *dependence traces*, and about existence of transparent runs for variables. And we need to ensure finiteness of the domain in order to ensure that fixpoint computation becomes effective. The latter problem is solved by introducing first, a subsumption order on dependence traces and, secondly, a notion of shortness of dependence traces. We then work with anti-chains (with respect to the subsumption order) of short dependence traces. It is not at all obvious that one can define on this abstract domain operations that are both sound and precise abstractions of the corresponding operations on non-atomic run sets. But we show in Chapter 4 that this is the case.

Detecting Copy Constants and Eliminating Faint Code

By solving the constraint system for bridging runs over the abstract domain introduced in Chapter 2 we can determine in particular the dependences mediated by bridging runs. As mentioned, this information can then be used to detect copy constants and eliminate faint code. Algorithms that solve these problems are given in Chapter 5 and their run-time is analyzed. These algorithms witness that we can detect copy constants and eliminate faint code in parallel programs, if we abandon the assumption that base statements execute atomically.

Intractability

The presented algorithms run in exponential time, which raises the question whether there are also efficient algorithms for these problems. In Chapter 6 we show by means of a reduction from the well-known SAT-problem that the answer is 'no', unless P=NP. Unlike the reductions in [26], this reduction relies only on active propagation along copying assignments but not on well-directed re-initialization. It applies independently of the atomicity assumption for base statements. In the conclusions, Chapter 7, we sketch possible remedies and discuss directions of future research that may still lead to algorithms of practical interest.

Interestingly, the programs constructed from SAT-instances are loop-free and do not employ procedures. Hence already for loop-free programs the analysis problems are intractable. For this class of programs it is rather easy to see that the problem can also be solved in non-deterministic polynomial time: witnessing runs can visit any program point at most once and can hence be guessed even in linear time. Therefore, for this class the problems are NP-complete.

Acknowledgment

I thank Helmut Seidl for many discussions that stimulated the research reported here and helped to clarify subtle points. He invited me to work for half a year in his research group at the University of Trier which allowed me to elaborate these ideas. Moreover, I am indebted to Jens Knoop, Oliver Rüthing, and Bernhard Steffen who shared part of their rich knowledge of data-flow analysis with me.

The research reported here was partially supported by the RTD project IST-1999-20527 "DAEDALUS" of the European FP5 programme.

Chapter 2

Parallel Flow Graphs

In this chapter, we introduce a flow graph model for parallel programs (cf. [33, 21, 13]). Edges in the flow graph are annotated with a base statement, a call of a single procedure, or a parallel call of two procedures. As base statements we allow assignment statements and the do-nothing statement ***skip***. We assume that branching is non-deterministic, a common abstraction in flow analysis.

There are two reasons for using a flow graph model instead of syntactic programs. Firstly, the nodes of a flow graph directly correspond to program points. Thus, they provide a natural entity to associate data flow information with in the analysis algorithm we are going to develop. Secondly, a flow graph model is slightly more general than a syntactic program model, as it also covers programs with unstructured control flow. This makes the positive results shown in this report (decidability of various analysis problems) slightly more general.

2.1 Parallel Flow Graphs

Let X be a finite set of *program variables* and Expr a set of expressions (or terms) over X. The precise nature of expressions is immaterial for the moment; we only need that each variable $x \in X$ is also an expression: $X \subseteq \mathsf{Expr}$, and that we can determine for an expression $t \in \mathsf{Expr}$ the set of variables occurring in t, $\mathsf{var}(t) \subseteq X$. Let $\mathsf{Stmt} := \{x := t \mid x \in X, t \in \mathsf{Expr}\} \cup \{\boldsymbol{skip}\}$ be the set of *base statements*. We use *stmt* to range over base statements.

Formally, a *parallel flow graph* comprises a finite set Proc of *procedure names* that contains a distinguished procedure *Main*. Intuitively, *Main* is the procedure with which execution starts. Each procedure name $p \in \mathsf{Proc}$ is associated with a control flow graph $G_p = (N_p, E_p, A_p, e_p, r_p)$ that consists of:

- a set N_p of *program points*;
- a set of edges $E_p \subseteq N_p \times N_p$;

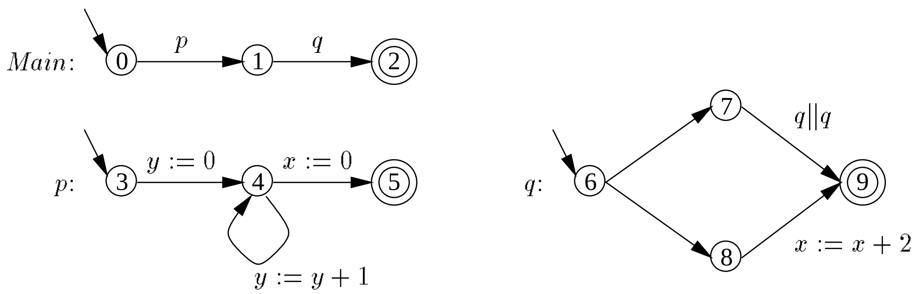


Figure 2.1: An example of a parallel flow graph.

- a mapping $A_p : E_p \to \mathsf{Stmt} \cup \mathsf{Proc} \cup \mathsf{Proc}^2$ that annotates each edge with a base statement, a call of a single procedure, or a parallel call of two procedures; and
- a special *entry (or start) point* $e_p \in N_p$ and a special *return point* $r_p \in N_p$.

We assume that the program points of different procedure are disjoint: $N_p \cap N_q = \emptyset$ for $p \neq q$. This can always be enforced by renaming program points.

We write N for $\bigcup_{p \in \mathsf{Proc}} N_p$, E for $\bigcup_{p \in \mathsf{Proc}} E_p$, and A for $\bigcup_{p \in \mathsf{Proc}} A_p$. We also agree that $\mathsf{Base} = \{e \mid A(e) \in \mathsf{Stmt}\}$ is the set of base edges, $\mathsf{Call}_p = \{e \mid A(e) = p\}$ is the set of edges that call procedure p, and $\mathsf{Pcall}_{p,q} = \{e \mid A(e) = (p, q)\}$ is the set of edges that call procedure p and q in parallel. Moreover, we write Call for $\bigcup_{p \in \mathsf{Proc}} \mathsf{Call}_p$ and Pcall for $\bigcup_{p,q \in \mathsf{Proc}} \mathsf{Pcall}_{p,q}$.

Example 2.1 *Figure 2.1 shows an example parallel flow graph with three procedures, Main, p, and q. The entry state of each procedure is marked by an arrow and the return state is indicated by a doubly circled state. The edge annotation* ***skip*** *is suppressed for clarity.*

The main procedure of the example flow graph sequentially starts procedures p and q. Procedure p sets variable y to an arbitrary non-negative value and initializes x by 0. Procedure q has a choice: it can execute either the upper path, where it starts two new instances of q in parallel or the lower path, where it increments x by 2. Note that arbitrarily many instances of q can run in parallel. Upon termination y can hold an arbitrary non-negative number and x can hold an arbitrary non-negative number that is even.

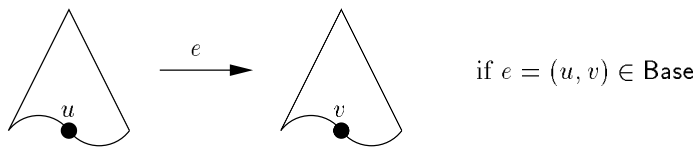
The example is intended to illustrate all the entities present in a parallel flow graph rather than to be an interesting example of a parallel program. □

The purpose of the remainder of this chapter is to set up a number of constraint systems, the solutions of which capture certain run sets. In the next section we define an operational semantics that is useful as a reference point for setting up these constraint systems correctly.

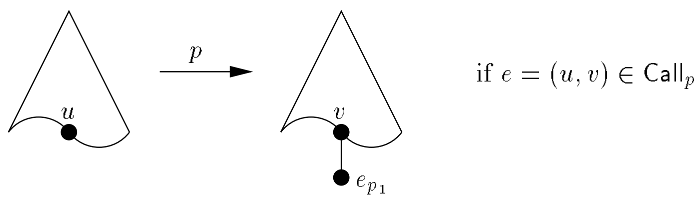
2.2 Operational Semantics

In the sequential interprocedural case, configurations of the operational semantics are given by sequences of flow graph nodes. Each sequence models a stack of return addresses (or rather return nodes). In parallel flow graphs procedures can also be called in parallel. We can model this by generalizing configurations from sequences to trees. Each node of the tree is labeled by a flow graph node. Each inner node of the tree has either degree one—such nodes correspond to return addresses from simple calls or to return addresses from parallel calls where one of the parallel threads has terminated already—or degree two—such nodes correspond to return addresses from parallel calls. The active control points are given by the leaves of the tree. Correspondingly, transitions are induced by the leaves. Transitions are labeled by base edges e, procedure names p, pairs of procedure names $p_0 \| p_1$, or the symbol ret. There are four transition rules:

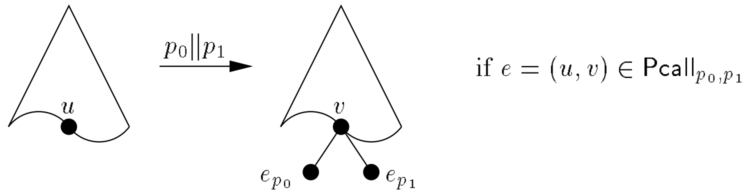
Base steps: $c \xrightarrow{e} c'$, if $e = (u, v) \in \mathsf{Base}$ and c' results from c by replacing a leaf labeled u by a leaf annotated with v.



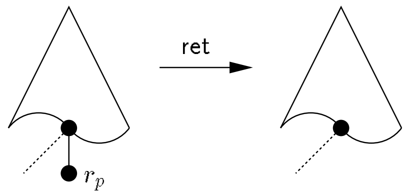
Simple calls: $c \xrightarrow{p} c'$, if there is an edge $e = (u, v) \in \mathsf{Call}_p$ such that c' results from c by replacing a leaf labeled u by a tree consisting of two nodes, a root labeled v and a successor node of the root labeled e_p.



Parallel calls: $c \xrightarrow{p_0 \| p_1} c'$, if there is an edge $e = (u, v) \in \mathsf{Call}_{p_0, p_1}$ such that c' results from c by replacing a leaf labeled u by a tree consisting of three nodes, a root labeled v with two successor nodes labeled e_{p_0} and e_{p_1}.



Return: $c \xrightarrow{\mathsf{ret}} c'$, if c' results from c by removing a leaf labeled by r_p for some $p \in \mathsf{Proc}$.



Note that the father of the node labeled r_p may become a leaf after application of this rule and may thus become active again. This models a return to the stacked return address. Just as well, however, the father of the node labeled r_p may also still have a child if it had degree two in c as indicated by the dotted line in the picture. In this case the father becomes active only after the second leaf also vanishes. This models synchronized termination of threads started by a parallel call.

Note also that the application of this rule to a tree consisting of just a root results in the empty tree. Such a step models overall termination.

Let Conf be the set of configurations, i.e., trees the degree of which is bounded by two and in which each node is annotated by a program point $u \in N$. We identify each program point $u \in N$ with the tree consisting of just a root labeled with u. We also write nil for the empty tree. A program point $u \in N$ is *active* in a configuration c, if it labels one of the leaves of c. The predicate $At_u(c)$ is true if u is active in c and false otherwise.

Let $\mathsf{Label} = N \cup \mathsf{Proc} \cup \mathsf{Proc}^2 \cup \{\mathsf{ret}\}$ be the set of transition labels and $\longrightarrow \subseteq \mathsf{Conf} \times \mathsf{Label} \times \mathsf{Conf}$ be the transition relation defined by the rules above. We define the transitive generalization $\Longrightarrow \subseteq \mathsf{Conf} \times \mathsf{Label}^* \times \mathsf{Conf}$ of $\longrightarrow$, by

$$\stackrel{\varepsilon}{\Longrightarrow} = \mathsf{Id} \qquad\qquad \stackrel{r \cdot l}{\Longrightarrow} = \stackrel{r}{\Longrightarrow}; \stackrel{l}{\longrightarrow},$$

where ';' denotes relational composition, and write $\Longrightarrow$ for $\cup_{r \in \mathsf{Label}^*} \stackrel{r}{\Longrightarrow}$.

2.3 Atomic Runs

A sequence of base edges is called an *(atomic) run*. Correspondingly, the set of atomic runs is $\mathsf{Runs} = \mathsf{Base}^*$. The classification 'atomic' refers to the fact that flow graph edges constitute atomic entities of execution; in Chapter 3 we will also consider 'non-atomic runs' at length. We define for a label sequence l, $\hat{l}$ to be the run obtained from l by retaining just the base edges and removing everything else:

$$\hat{\varepsilon} = \varepsilon \quad \text{and} \quad \widehat{rl} = \begin{cases} \hat{r}l & \text{if } l \in \mathsf{Base} \\ \hat{r} & \text{otherwise} \end{cases} \qquad \text{for } r \in \mathsf{Label}^*,\ l \in \mathsf{Label}\,.$$

In the following we are going to set up constraint systems for a variety of run sets. These constraint systems use the following small number of operators and constants on run sets.

Semantics of base edges: $[\![e]\!] = \{\langle e \rangle\}$ for $e \in \mathsf{Base}$. This characterizes the run induced by a base edge in isolation.

Sequential composition operator: $R\,;S = \{rs \mid r \in R, s \in S\}$. This characterizes the sequential composition of run sets.

Interleaving operator: In order to define the interleaving (or parallel composition) operator some notation is needed. Let $r = \langle e_1, \ldots, e_n \rangle$ be a sequence and $I = \{i_1, \ldots, i_k\}$ a subset of positions in r such that $1 \leq i_1 < i_2 < \cdots < i_k \leq n$. Then $r|I$ is the sequence $\langle e_{i_1}, \ldots, e_{i_k} \rangle$. We write $|r|$ for the length of r, viz. n.

Then the interleaving of R and S is defined by

$$\begin{aligned} R \otimes S \;\; = \;\; \{r \mid \exists I_R, I_S : \; & I_R \cup I_S = \{1, \ldots, |r|\}, I_R \cap I_S = \emptyset, \\ & r|I_R \in R, r|I_S \in S, \}\,. \end{aligned}$$

Prefix operator: $pre(R) = \{r \mid \exists s : rs \in R\}$. This captures prefixes of the runs of R.

Postfix operator: $post(R) = \{r \mid \exists s : sr \in R\}$. This captures postfixes of the runs in R.

Alternatively, atomic runs may be defined as sequences of base statements instead of base edges. The only thing that needs to be changed is to redefine Runs as Stmt^* instead of Base^* and to redefine $[\![e]\!]$ by $[\![e]\!] = \{\langle A(e) \rangle\}$. The other operators are not affected by this change. In this setting we should also redefine the hat-operator to incorporate the transition from base edges to base statements:

$$\hat{\varepsilon} = \varepsilon \quad \text{and} \quad \widehat{rl} = \begin{cases} \hat{r} \cdot A(l) & \text{if } l \in \mathsf{Base} \\ \hat{r} & \text{otherwise} \end{cases} \qquad \text{for } r \in \mathsf{Label}^*,\, l \in \mathsf{Label}\,.$$

The remainder of this chapter can be read with both interpretations.

By redefining the operators on run sets, we can obtain non-standard semantics. On the one hand, this is used in Chapter 3 for defining a semantics for parallel flow graphs in which execution of base edges is no longer assumed to be atomic. On the other hand, we can redefine these operators on an abstract domain with a finite chain height. Over such a domain we can effectively solve the constraint systems to be introduced soon. If we can show that all operators are correct or even precise abstractions of the concrete operators on atomic or non-atomic run sets, standard abstraction theorems from abstract interpretation ensure that the solution we get is a correct or even precise abstraction of the run sets characterized by the constraint systems. This is the idea of constraint-based program analysis.

2.4 The Run Sets of Ultimate Interest

We are ultimately interested in setting up constraint systems that characterize for each $u \in N$ the following sets of runs:

Reaching runs: $\mathsf{R}(u) = \{\hat{r} \mid e_{Main} \stackrel{r}{\Longrightarrow} c, At_u(c)\}$.

Terminating runs: $\mathsf{T}(u) = \{\hat{r} \mid e_{Main} \Longrightarrow c \stackrel{r}{\Longrightarrow} \mathsf{nil}, At_u(c)\}$.

In data-flow analysis one considers *forward-* and *backward-*analyses. Forward-analyses calculate abstractions of the reaching runs and backward analyses abstractions of the terminating runs.

We are also interested for all program points $u, v \in N$ in the set of those runs that potentially transfer information from u to v. We call these the *bridging runs from u to v*, or *u-v-runs* for short.

Bridging runs: $\mathsf{B}_v(u) = \{\hat{r} \mid e_{Main} \Longrightarrow c_u \stackrel{r}{\Longrightarrow} c_v, At_u(c_u), At_v(c_v)\}$.

In order to characterize these run sets as smallest solutions of constraint systems, auxiliary run sets are necessary. In the sections that follow we stepwise introduce these auxiliary run sets and present constraint systems that characterize them. We always explain the underlying intuition but leave the details of the proof that the presented constraint systems capture the described run system to the reader. The constraint systems for same-level, reaching and terminating runs are essentially taken from [33] where, however, they are not justified with reference to an explicitly stated underlying operational semantics. The constraint system for bridging runs is new.

2.5 The Constraint Systems

2.5.1 Same-Level Runs

First of all, we characterize so-called *same-level runs*. Same-level runs of procedures capture complete runs of procedures in isolation.

Same-level runs of procedures: $\mathsf{S}(q) = \{\hat{r} \mid e_q \stackrel{r}{\Longrightarrow} \mathsf{nil}\}$ for $q \in \mathsf{Proc}$.

As auxiliary sets we consider same-level runs to program nodes.

Same-level runs to program nodes: $\mathsf{S}(u) = \{\hat{r} \mid e_q \stackrel{r}{\Longrightarrow} u\}$ for $u \in N_q$, $q \in \mathsf{Proc}$.

Same-level runs of procedures form an important building block for the other constraint systems. They play a similar role to summary edges in interprocedural program analysis:[1] the same-level runs of procedure q summarize the complete effect of call edges $e \in \mathsf{Call}_p$. Also the complete effect of a parallel call edge $e \in \mathsf{Pcall}_{p_0,p_1}$ is obtained easily from the same-level runs of procedures p_0 and p_1: it is given by $\mathsf{S}(p_0) \otimes \mathsf{S}(p_1)$.

The same-level runs of procedures and program nodes are the smallest solution of the following constraint system:

$$
\begin{array}{lll}
[\mathrm{S1}] & S(q) \supseteq S(r_q) & \\
[\mathrm{S2}] & S(e_q) \supseteq \{\varepsilon\} & \\
[\mathrm{S3}] & S(v) \supseteq S(u) \, ; [\![e]\!] \, , & \text{if } e = (u,v) \in \mathsf{Base} \\
[\mathrm{S4}] & S(v) \supseteq S(u) \, ; S(p) \, , & \text{if } e = (u,v) \in \mathsf{Call}_p \\
[\mathrm{S5}] & S(v) \supseteq S(u) \, ; [S(p_0) \otimes S(p_1)] \, , & \text{if } e = (u,v) \in \mathsf{Pcall}_{p_0,p_1}
\end{array}
$$

It is easy to see that the same-level runs satisfy all constraints:

[S1]: A same-level run of the return point of procedure q gives rise to a same-level run of q by the Return Rule.

[S2]: It follows trivially from the definition that ε is a same-level run of the entry point of a procedure.

[S3]: If $e = (u, v)$ is a base edge, we get a same-level run to v by extending a same-level run to u with e by the Base Steps Rule.

[S4]: If $e = (u, v)$ is an edge that calls p, we get a same-level run to v if we extend a same-level run to u by a same-level run of p: we follow the execution underlying the same-level run to v and apply then call p according to the Call Rule; we then follow the execution underlying the same-level run of p (with v waiting on the stack to become active) and return to v according to the Return Rule.

[S5]: Similarly, if $e = (u, v)$ is an edge that calls p_0 and p_1 in parallel, we can—after seeing a same-level run to u—follow this edge; then p_0 and p_1 are performed to completion in parallel, which results in an interleaving of a same-level run of p_0 and p_1; after that execution returns to v. We thus obtain a same-level run to v by extending a same-level run of u with an interleaving of same-level runs of p_0 and p_1.

On the other hand, we can easily prove by induction on the length of the transition sequences inducing same-level runs, that each same-level run lies in the smallest solution of the constraint system: in the base case we consider the empty

[1] Indeed, the information associated with a summary edge for p usually is an abstraction of the same-level runs of p.

execution ε. It can only give rise to the same-level run ε to e_q for some procedure q. But ε is enforced to lie in the solution of $S(r_p)$ explicitly by constraint [S2].

In the induction step, we consider longer executions leading to same-level runs. The execution underlying a same-level run of a procedure q necessarily involves a final return from r_q after an execution that gives rise to a same-level run of r_q. The latter execution is one step shorter and thus the same-level run of r_q is contained in the solution of $S(r_q)$ by the induction hypothesis. Now, the constraint [S1] ensures that it is also contained in the set assigned to $S(q)$ in a solution.

The last step of a non-empty execution r inducing a same-level run $\hat{r}$ to a program point v must be induced either by the Base Rule or the Return Rule because the simple and parallel call rules never lead to a configuration which consists of just a single state. If the last step is induced by the Base Rule, the previous configuration is a program point u. Then $\hat{r}$ is composed of a same-level run to u and the base edge $e = (u, v)$. The same-level run to u is induced by a shorter execution an hence contained in the set associated with $S(u)$ in a solution by the induction hypothesis. Thus, $\hat{r}$ is in $S(v)$ by the constraint [S3]. If the last step is induced by the Return Rule, then there must be a simple or parallel call from which this step returns. The constraints for simple and parallel call edges ([S4] and [S5]) together with the induction hypothesis then ensure that $\hat{r}$ is contained in $S(v)$.

2.5.2 Inverse Same-Level Runs

We also consider a kind of dual to same-level runs of program points: runs from a program point to the return point of the corresponding procedure. We call these *inverse same-level runs of program point.* They are needed in order to capture terminating runs.

Inverse same-level runs of program points: $\mathsf{S}^{\mathsf{i}}(u) = \{\hat{r} \mid u \stackrel{r}{\Longrightarrow} \mathsf{nil}\}$ for $u \in N$.

Inverse same-level runs of procedures and program nodes are obtained by backwards accumulation as the smallest solution of the following system of constraints:

$$
\begin{array}{lll}
[\text{SI1}] & S^{\mathsf{i}}(r_q) \supseteq \{\varepsilon\} & \\
[\text{SI2}] & S^{\mathsf{i}}(u) \supseteq [\![e]\!]\,; S^{\mathsf{i}}(v)\,, & \text{if } e = (u, v) \in \mathsf{Base} \\
[\text{SI3}] & S^{\mathsf{i}}(u) \supseteq S(p)\,; S^{\mathsf{i}}(v)\,, & \text{if } e = (u, v) \in \mathsf{Call}_p \\
[\text{SI4}] & S^{\mathsf{i}}(u) \supseteq [S(p_0) \otimes S(p_1)]\,; S^{\mathsf{i}}(v)\,, & \text{if } e = (u, v) \in \mathsf{Pcall}_{p_0, p_1}
\end{array}
$$

The last two constraints refer to same-level runs of procedures. Therefore, it appears that we need to calculate same-level runs before we can calculate inverse same-level runs by the above constraint system. However, by adding for each procedure $q \in \mathsf{Proc}$ the constraint

$$[\text{SI5}] \quad S(q) \supseteq S^{\mathrm{i}}(e_q)$$

we can calculate same-level runs of procedures simultaneously with inverse same-level runs. Thus, we can also calculate inverse same-level runs in isolation if we wish to do so.

It is easy to see that the sets of inverse same-level runs satisfy all constraints:

[SI1]: By the Return rule, ε clearly is an inverse same-level run of the return point r_q of a procedure.

[SI2]: If $e = (u, v)$ is a base edge, we get an inverse same-level run of u by prefixing a same-level run of v with e.

[SI3]: If $e = (u, v)$ is an edge that calls p, we can follow this edge in an execution from u; then first p is performed until termination, which results in a same-level run of p; after that execution proceeds at v. We thus obtain an inverse same-level run of u by prefixing an inverse same-level run of v by a same-level run of p.

[SI4]: Similarly, if $e = (u, v)$ is an edge that calls p_0 and p_1 in parallel, we can follow this edge in an execution from u; then first p_0 and p_1 are performed to completion in parallel, which results in an interleaving of a same-level run of p_0 and p_1; after that execution returns to v. We thus obtain an inverse same-level run of u by prefixing an inverse same-level run of v with an interleaving of same-level runs of p_0 and p_1.

On the other hand, we can easily prove by induction on the length of the transition sequences inducing inverse same-level runs, i.e. those that lead to nil, that each inverse same-level run is in the smallest solution of the constraint system: in the base case we consider the shortest executions that lead to same-level runs. These are executions of the form $r_p \xrightarrow{\mathsf{ret}} \mathsf{nil}$ for some procedure p. They witness that $\varepsilon \in \mathsf{S}(r_p)$. But ε is enforced to be in a solution of $S(r_p)$ explicitly by constraint [SI1].

In the induction step, we consider longer executions leading to same-level runs. These necessarily start with a transition induced by a base edge, a simple, or a parallel call edge. The resulting run is then composed from shorter runs as specified in the constraints for base edges ([SI2]), simple calls ([SI3]), and parallel calls ([SI4]), respectively.

2.5.3 Two Assumptions and a Simple Analysis

The following two assumptions simplify the constraint systems that follow:

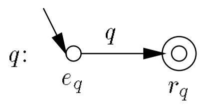
ASS1: every program point $u \in N_q$ in a procedure q can be reached by a same-level run from the entry point e_q of q:

$$\forall q \in \mathsf{Proc}, u \in N_q : \mathsf{S}(u) \neq \emptyset\,.$$

ASS2: from every program point $u \in N_q$ the return point r_q can be reached by a same-level run:

$$\forall q \in \mathsf{Proc}, u \in N_q : \mathsf{S}^{\mathsf{i}}(u) \neq \emptyset\,.$$

These assumptions are not as innocent as it may seem at first glance. In particular it does *not* suffice to require that there are paths from e_q to u and from u to r_q in the flow graph G_q for q. This is just a necessary but not a sufficient condition. The paradigmatic counter-example is a procedure that calls itself:



Although there is a path from e_q to r_q in the flow graph, no execution can reach r_q from e_q, as there is no terminating bypass of the recursive call of q. Hence both $\mathsf{S}(r_q)$ and $\mathsf{S}^{\mathsf{i}}(e_q)$ are empty. Examples like this show that we cannot assume without loss of generality that practical flow graphs satisfy ASS1 and ASS2.

While assumptions ASS1 and ASS2 simplify the presentation and justification of the constraint systems in the remainder of this chapter, they are not strictly necessary. We can well design constraint systems that work in the general case, but they are more complex and therefore harder to explain. In order to avoid overloading the presentation we decided for a two-level approach, where we first present and justify the simpler constraint systems that work if ASS1 and ASS2 are satisfied. Afterwards we explain which changes are necessary for the general case, cf. Section 2.5.7.

In order to compute the information needed to decide ASS1 and ASS2, we design a simple analysis procedure based on an abstract interpretation of the operators and constants used in the constraint systems. The information computed by this analysis is also used for setting up the constraint systems for the general case. We work with a two point domain ($\mathbb{D} = \{\perp, \top\}, \leq$) ordered as $\perp \leq \top$. The idea is that $\perp$ represents emptiness of a run set and $\top$ non-emptiness. This is formally captured by the abstraction mapping $\alpha : 2^{\mathsf{Runs}} \to \mathbb{D}$, defined by $\alpha(\emptyset) = \perp$ and $\alpha(R) = \top$ for $R \neq \emptyset$. Obviously, α is universally disjunctive. We define the abstract interpretation of the operators by

$$x ;^{\#} y = x \otimes^{\#} y = x \wedge y\,, \quad pre^{\#}(x) = post^{\#}(x) = x\,, \quad [\![e]\!]^{\#} = \{\varepsilon\}^{\#} = \top$$

for $x, y \in \mathbb{D}$, $e \in E$. It is easy to see that the abstract operators are precise abstractions of the corresponding operators on run sets: a sequential or parallel composition of two run sets is non-empty, iff both arguments are non-empty; the set of prefixes and the set of postfixes of a run set R are non-empty iff R is; and each base edge gives rise to a non-empty run set. In other words, α is a strong homomorphism. Therefore, by solving the constraint systems for same-level and inverse same-level runs over the abstract interpretation we get precise information about the emptiness of the sets of same-level and inverse same-level runs of program points.

This analysis is cheap: as $(\mathbb{D}, \leq)$ has chain height two the information for each constraint variable can change at most once in the fixpoint iteration. By standard demand-driven fixpoint evaluation, we can organize the fixpoint computation such that each operator in the constraint system is evaluated at most once. Thus, the computation can be done in time $O(|E| + |\mathsf{Proc}|)$, the number of operators in the constraint systems. As in all practical flow graphs out-degrees of program nodes are bounded, typically by 2 and $|\mathsf{Proc}|$ is trivially bounded by $|N|$ as each procedure has a distinguished entry node, this is $O(|N|)$. In the following we assume that this analysis has been done such that for each program node u and procedure q the information whether $\mathsf{S}(u)$, $\mathsf{S}^{\mathsf{i}}(u)$, $\mathsf{S}(q)$, or $\mathsf{S}^{\mathsf{i}}(q)$ is empty or not is readily available.

Another analysis that can determine information about reachability of program points in parallel flow graphs has been described by Seidl and Steffen [33] as an instance of their generic analysis framework for solving gen/kill data flow problems for parallel programs.

2.5.4 Reaching Runs

As auxiliary sets for characterizing the runs that reach a program point u, we consider the runs that reach u from a call to procedure q.

Reaching runs from procedures: $\mathsf{R}(u,q) = \{\hat{r} \mid e_q \stackrel{r}{\Longrightarrow} c, At_u(c)\}$ for $u \in N$, $q \in \mathsf{Proc}$.

With this definition, we obviously have $\mathsf{R}(u) = \mathsf{R}(u, Main)$. Hence we are done with characterizing reaching runs if we succeed in characterizing reaching runs from procedures. The latter can be characterized as the smallest solution of the following constraint system:

[R1]	$R(u,q) \supseteq S(u)\,,$	if $u \in N_q$
[R2]	$R(u,q) \supseteq S(v)\,; R(u,p)\,,$	if $(v,\_) \in E_q \cap \mathsf{Call}_p$
[R3]	$R(u,q) \supseteq S(v)\,; [R(u,p_i) \otimes pre(S(p_{1-i}))]\,,$	if $(v,\_) \in E_q \cap \mathsf{Pcall}_{p_0,p_1}$

The last clause is meant to specify two constraint for $i = 0$ and $i = 1$.

The reaching runs satisfy the constraints:

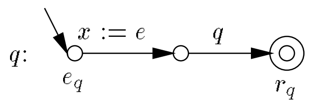
[R1]: Firstly, each same-level run of u clearly is also a reaching run of u.

[R2]: Secondly, if we have a program point v in q that has an outgoing edge calling p—the situation described in the second constraint—we obtain a run that reaches u from q when we extend a same-level run $\hat{r}$ to v with a run $\hat{r'}$ that reaches u from p (where r and r' are the underlying executions). The corresponding execution is this: first we follow r; this brings us to v where we call p; we then follow r' (with the target node of the edge $(v, \_) \in \mathsf{Call}_p$ on the stack).

[R3]: Thirdly, consider a program point v in q that has an outgoing edge calling p_0 and p_1 in parallel, the situation described in the third constraint. Similar to the second case, we get a run reaching u by extending a same-level run of v with a run that reaches u in the parallel call. The latter can happen either in p_0 or p_1 hence the two cases with $i = 0, 1$. Now until p_i has reached u in p_i the other procedure p_{1-i} can perform a prefix of a same-level run.

On the other hand, the constraint system captures all the ways how u may be reached from e_q. There are just three possibilities: either u is on the same-level, in a simple call, or in a parallel call. These case are completely covered by the constraints.

Note that assumption ASS2 is crucial for making the constraint for parallel calls sufficiently rich. If it is violated, the partial run exhibited by p_{1-i} while p_i is in the process of reaching u must not be a prefix of a same-level run. For example, the following procedure q might execute $x := e$ arbitrarily often, although $S(q)$ and hence $pre(S(q))$ is empty.



A possible remedy is described in Section 2.5.7.

2.5.5 Terminating Runs

The approach for capturing terminating runs is dual to the ones for reaching runs. As auxiliary sets we consider terminating runs of u in a call to procedure q.

Terminating runs in procedures: $\mathsf{T}(u,q) = \{\hat{r} \mid e_q \Longrightarrow c \xRightarrow{r} \mathsf{nil}, At_u(c)\}$ for $u \in N$, $q \in \mathsf{Proc}$.

Obviously we have $\mathsf{T}(u) = \mathsf{T}(u, Main)$ such that it suffices to capture terminating runs in procedure calls in the constraint system. The constraint system is dual to the one for reaching runs:

[T1]	$T(u,q) \supseteq S^{\mathrm{i}}(u)\,,$	if $u \in N_q$
[T2]	$T(u,q) \supseteq T(u,p)\,;S^{\mathrm{i}}(w)\,,$	if $(\_, w) \in E_q \cap \mathsf{Call}_p$
[T3]	$T(u,q) \supseteq [T(u,p_i) \otimes post(S(p_{1-i}))]\,;S^{\mathrm{i}}(w)\,,$	if $(\_, w) \in E_q \cap \mathsf{Pcall}_{p_0,p_1}$

Again, $i = 0, 1$ in the last constraint. The justification of this constraint system is similar to reaching runs; therefore, the details are left to the reader. We should mention, however, that assumption ASS1 is crucial here, like ASS2 in the case of reaching run, but for a quite different reason. The fundamental difference is the requirement that the configuration c with $At_u(c)$ is reachable ($e_q \Longrightarrow c$) in terminating runs, a requirement that has no analogue for reaching runs. As a consequence, $post(\mathsf{S}(p_{1-i}))$ is now sufficient to capture the interleaving potential in the constraint for parallel calls even in the general case, in contrast to $pre(\mathsf{R}(p_{1-i}))$ in the corresponding constraint for reaching runs.

However, the reachability requirement for configuration c, implies that some of the constraints are not satisfied by the sets $\mathsf{T}(u, q)$ in the general case. For example, an inverse same-level run r from a program point $u \in N_q$ is not always a terminating run. Being an inverse same-level run just means that $u \stackrel{r}{\Longrightarrow} \mathsf{nil}$ holds, but for a terminating run we additionally need $e_q \Longrightarrow u$. This is automatically true if ASS1 is valid but can be wrong in the general case. Similarly, we need that the start node of the edge e in the second and third constraint can be reached for making the constraints valid for the operationally defined sets. A possible remedy is to remove the constraints induced by non-reachable program points. This is detailed in Section 2.5.7.

2.5.6 Bridging Runs

Let $v \in N$ be a fixed program point. We want to determine the bridging runs $\mathsf{B}_v(u)$ for each $u \in N$ as defined in Section 2.4. As a first step we capture for each program points u the runs that reach v, when execution is started directly with u. We call these the *simple bridging runs* of u w.r.t. v.

Simple bridging runs: $\mathsf{B}^{\mathsf{s}}_v(u) = \{\hat{r} \mid u \stackrel{r}{\Longrightarrow} c, At_v(c)\}$ for $u \in N$.

The simple bridging runs can be characterized as the smallest solution of the following constraint system:

$$
\begin{array}{lll}
[\mathrm{BS1}] & B^{\mathsf{s}}(v) \supseteq \{\varepsilon\} & \\
[\mathrm{BS2}] & B^{\mathsf{s}}(u) \supseteq [\![e]\!] \,; B^{\mathsf{s}}(w)\,, & \text{if } e = (u, w) \in \mathsf{Base} \\
[\mathrm{BS3}] & B^{\mathsf{s}}(u) \supseteq S(p) \,; B^{\mathsf{s}}(w)\,, & \text{if } e = (u, w) \in \mathsf{Call}_p \\
[\mathrm{BS4}] & B^{\mathsf{s}}(u) \supseteq B^{\mathsf{s}}(e_p)\,, & \text{if } e = (u, \_) \in \mathsf{Call}_p \\
[\mathrm{BS5}] & B^{\mathsf{s}}(u) \supseteq [S(p_0) \otimes S(p_1)] \,; B^{\mathsf{s}}(w)\,, & \text{if } e = (u, w) \in \mathsf{Pcall}_{p_0, p_1} \\
[\mathrm{BS6}] & B^{\mathsf{s}}(u) \supseteq B^{\mathsf{s}}(e_{p_i}) \otimes pre(S(p_{1-i}))\,, & \text{if } e = (u, \_) \in \mathsf{Pcall}_{p_0, p_1}
\end{array}
$$

The last constraint is again included for $i = 0, 1$.

Let us explain why these constraints cover all the ways how v can be reached from u. If $u = v$ then there is the trivial way to reach v from u: by the empty execution; this is covered by Constraint [BS1]. Otherwise, we must proceed via an outgoing edge (u, w) of u. If this is a base edge $e = (u, w)$, we first see e and then a run that reaches v from w; this is covered by Constraint [BS2]. If e is an edge that calls a procedure p, we distinguish two cases: either v is reached after p has terminated—this case is covered by Constraint [BS3]—or v is reached during the execution of p—this case is covered by the [BS4]. Similarly, if e is a parallel call of two procedures p_0 and p_1, we can reach v either after both procedures have terminated, which is covered by [BS5]. Or we can reach v in one of the called procedures p_i. In this case we see a run from e_{p_i} that reaches v interleaved with a prefix of a same-level run of procedure p_{1-i}. If assumption ASS2 is violated we must again reckon with procedure p_{1-i} providing runs that are not prefixes of same-level runs, as was the case for reaching runs. We can resolve this again by replacing $pre(S(p_{1-i}))$ by $\mathsf{P}(p_{1-i})$ in Constraint [BS6].

The reader should face no difficulties in persuading himself, that the $\mathsf{B}^{\mathsf{s}}(u)$ sets indeed solve all constraints.

As a second step we determine the bridging runs in a call to a procedure:

Bridging runs in procedure calls:

$$\mathsf{B}_v(u, q) = \{\hat{r} \mid e_q \Longrightarrow c_u \stackrel{r}{\Longrightarrow} c_v, At_u(c), At_v(c)\} \text{ for } u \in N.$$

Clearly, we have $\mathsf{B}_v(u) = \mathsf{B}_v(u, Main)$ such that we are done, when we have successfully captured $\mathsf{B}_v(u, q)$ for all u, q.

Basically, there are two ways how a bridging run may occur in a call to q. One possibility is that both u and v are reached in the same simple or parallel call in q. This case is captured by the following three types of constraints:

$$\begin{array}{lll}
[\mathrm{B1}] & B(u,q) \supseteq B(u,p)\,, & \text{if } e \in E_q \cap \mathsf{Call}_p \\
[\mathrm{B2}] & B(u,q) \supseteq B(u,p_i) \otimes post(pre(S(p_{1-i})))\,, & \text{if } e \in E_q \cap \mathsf{Pcall}_{p_0,p_1} \\
[\mathrm{B3}] & B(u,q) \supseteq pre(T(u,p_i)) \otimes post(R(v,p_{1-i}))\,, & \text{if } e \in E_q \cap \mathsf{Pcall}_{p_0,p_1}
\end{array}$$

[B1] and [B2] apply for $i = 0, 1$.

Constraint [B1] captures the case that u and v are reached in the same simple call. Constraint [B2] is concerned with the case that u and v are reached in the same procedure p_i of a parallel call. Before u is reached in p_i the other procedure can already perform certain actions and it need not run to completion until v is reached. Therefore, p_{1-i} contributes a middle piece of a same-level run. Potential middle pieces can be characterized by $pre(post(S(p_{1-i}))$ as captured by the second constraint. Constraint [B3] captures the case that u is reached in procedure p_i and v in procedure p_{1-i}. After p_i has reached u it can further proceed; specifically p_i contributes a prefix of a run from $\mathsf{T}(u)$ until v is reached in p_{1-i}. In order to reach v, p_{1-i} must execute a run from $\mathsf{R}(v, p_{1-i})$. It can execute a prefix of this run before p_i leaves u. Therefore, we see a postfix of a run from $\mathsf{R}(v, p_{1-i})$ as part of the bridging run.

The second possibility is that u and v are not reached in the same simple or parallel call. This gives rise to the following constraints:

[B4]	$B(u,q) \supseteq B^{\mathrm{s}}(u)\,,$	if $u \in N_q$
[B5]	$B(u,q) \supseteq T(u,p)\,;B^{\mathrm{s}}(w)\,,$	if $(\_,w) \in E_q \cap \mathsf{Call}_p$
[B6]	$B(u,q) \supseteq [T(u,p_i) \otimes \mathit{post}(S(p_{1-i}))]\,;B^{\mathrm{s}}(w)\,,$	if $(\_,w) \in E_q \cap \mathsf{Pcall}_{p_0,p_1}$

where $i = 0,1$ in the last constraint.

The first subcase is that u is reached 'on same-level', i.e. in the current instance of q. Then we see a simple bridging run of u (Constraint [B4]). The second subcase is that u is reached in a procedure p called by a simple call edge $e = (\_,v) \in E_q$. Then we see a run from $T(u,p)$ followed by a simple bridging run from w (Constraint [B5]). The third subcase is that u is reached in a procedure p_i called by a parallel call edge $e = (\_,v) \in E_q$. Then we see a run from $T(u,p_i) \otimes \mathit{post}(S(p_{1-i}))$ followed by a simple bridging run from w (Constraint [B6]).

2.5.7 The General Case

In this section we describe the changes that are necessary in the general case, i.e., if assumptions ASS1 and ASS2 are potentially violated.

As explained in connection with constraint [R3] one of the problems is that in the general case $\mathit{pre}(S(q))$ does not capture all partial runs of procedure q. Thus, interleaving $R(u,p_i)$ with $\mathit{pre}(S(p_{1-i}))$ does not capture all possible run that reach u in a parallel call. This problem also arises in constraints [BS6] and [B2]. A possible remedy is to introduce new variables $P(q)$, $q \in \mathsf{Proc}$, that characterize finite prefixes of (finite or infinite) runs, i.e. $\mathsf{P}(q) = \{\hat{r} \mid e_q \overset{r}{\Longrightarrow} c\}$, and to use $P(p_{1-i})$ instead of $\mathit{pre}(S(p_{1-i}))$ in [R3], [BS6], and [B2]. A simple way to calculate $\mathsf{P}(q)$ is to add a constraint of the following form for each procedure q and program point u to the constraint system for reaching runs:[2]

$$\text{[P]} \quad P(q) \supseteq R(u,q)\,.$$

While this way of calculating $\mathsf{P}(q)$ is easy to specify it has the disadvantage of introducing $|N| \cdot |\mathsf{Proc}|$ new constraints, i.e. quadratically many. Although this

[2] If we are working with a non-atomic interpretation of assignments we must use the following constraint instead of [P]:

$$\text{[P}'\text{]} \quad P(q) \supseteq \mathit{pre}(R(u,q))\,.$$

In the atomic interpretation, any configuration c satisfies $At_u(c)$ for at least one program point u. Therefore, the simpler constraint [P] without the pre-operator is sufficient. In the non-atomic interpretation, however, there are (implicitly) *transient* configurations that correspond to intermediate stages of executions in which no program point is active. Fortunately, from all transient configurations c a configuration c' with some active program point is reachable. Therefore, we can capture the runs to transient configurations by means of the pre-operator.

$$
\begin{array}{lll}
[\mathrm{P1}] & P(q) \supseteq P(e_q) & \\
[\mathrm{P2}] & P(u) \supseteq \{\varepsilon\} & \\
[\mathrm{P3}] & P(u) \supseteq [\![e]\!]\,;P(v)\,, & \text{if } e=(u,v) \in \mathsf{Base} \\
[\mathrm{P4}] & P(u) \supseteq P(p)\,, & \text{if } (u,\_) \in \mathsf{Call}_p \\
[\mathrm{P5}] & P(u) \supseteq S(p)\,;P(v)\,, & \text{if } (u,v) \in \mathsf{Call}_p \\
[\mathrm{P6}] & P(u) \supseteq [P(p_0) \otimes P(p_1)]\,, & \text{if } (u,v) \in \mathsf{Pcall}_{p_0,p_1} \\
[\mathrm{P7}] & P(u) \supseteq [S(p_0) \otimes S(p_1)]\,;S^{\mathrm{i}}(v)\,, & \text{if } (u,v) \in \mathsf{Pcall}_{p_0,p_1}
\end{array}
$$

Figure 2.2: A constraint system characterizing finite prefixes.

$$
\begin{array}{lll}
[\mathrm{Q1}] & Q(u,q) \supseteq P(u)\,, & \text{if } u \in S_q \\
[\mathrm{Q2}] & Q(u,q) \supseteq Q(u,p)\,, & \text{if } (v,\_) \in E_q \cap \mathsf{Call}_p \\
[\mathrm{Q3}] & Q(u,q) \supseteq T(u,p)\,;P(w)\,, & \text{if } (v,w) \in E_q \cap \mathsf{Call}_p \\
[\mathrm{Q4}] & Q(u,q) \supseteq Q(u,p_i) \otimes \mathit{post}(P(p_{1-i}))\,, & \text{if } (v,\_) \in E_q \cap \mathsf{Call}_{p_0,p_1} \\
[\mathrm{Q5}] & Q(u,q) \supseteq [T(u,p_i) \otimes \mathit{post}(S(p_{1-i}))]\,;P(w)\,, & \text{if } (v,w) \in E_q \cap \mathsf{Call}_{p_0,p_1}
\end{array}
$$

Figure 2.3: A constraint system for partial runs that can be exhibited in a procedure after a given program point has been reached. All constraints [Q1]-[Q5] are only for program points v with $\mathsf{S}(v) \neq \emptyset$. In [Q4] and [Q5], $i = 0, 1$.

does not spoil the overall asymptotic complexity—already the constraint system for reaching runs has $O(|N| \cdot |\mathsf{Proc}|)$ constraints—we should mention that $\mathsf{P}(q)$ can be calculated also by $O(|N|)$ constraints. A corresponding constraint system is given in Fig. 2.2. It determines as auxiliary information finite prefixes of (finite or infinite) runs from program points, defined by $\mathsf{P}(u) = \{\hat{r} \mid u \stackrel{r}{\Longrightarrow} c\}$ by backwards accumulation and is similar to the constraint system for simple bridging runs.

A similar problem arises in constraint [B3]: if assumption ASS2 is violated, $\mathit{pre}(T(u,p_i))$ does not necessarily capture all partial runs exhibited by p_i after reaching u because u could be reached at a configuration from which termination is impossible. The information needed in place of $\mathit{pre}(T(u,p_i))$ is $\mathsf{Q}(u,p_i)$ where $\mathsf{Q}(u,q) = \{\hat{r} \mid e_q \Longrightarrow c \stackrel{r}{\Longrightarrow} c', At_u(c)\}$ for $u \in N$, $q \in \mathsf{Proc}$. These sets can be characterized by the constraint system in Fig. 2.3

The above changes ensure that the run sets characterized by the constraint systems are sufficiently large. They are necessary to make flow analysis based on abstract interpretation of the constraint systems sound. The changes described now ensure that the run sets do not become too large. Thus, they are necessary to make analyses based on a precise abstract interpretation complete.

As explained in connection with terminating runs, constraints induced by unreachable program points are not satisfied by the run sets (defined from the operational semantics) that we intend to characterize. As these constraints pose unnecessary additional requirements they make the solutions larger than necessary. Fortunately, such constraints are also unnecessary for soundness and can simply be removed. Specifically, we must include the constraints [T1], [B1], and [B4] only for program points u with $\mathsf{S}(u) \neq \emptyset$, and the constraints [T2], [T3], [B2], [B3], [B5], and [B6] only for edges $e = (v, w)$ with $\mathsf{S}(v) \neq \emptyset$. We have seen in Section 2.5.3, that we can determine this information with a very simple analysis.

With the changes described in this section we obtain constraint systems that are both sound and complete in the general case.

2.6 Discussion

In this chapter we have introduced parallel flow graphs. After that we defined a symbolic operational semantics. It works on configurations that take the form of a tree, the nodes of which are annotated by program points. Intuitively, such a tree models a generalization of a run time stack that may branch to parallel stacks in addition to the common stack operations. Branching is crucial to model parallel calls. We have described the transitions of the operational semantics by rules that work directly on configurations of this form.

There are obvious alternatives to this way of describing the operational semantics: in particular, we could have used the approach chosen by Esparza, Knoop, and Podelski in their work on flow analysis of parallel programs [9, 10]. They map a parallel flow graph to a so-called PA-processes; PA is a process algebra which has both a sequential and a concurrent composition operator [3, 22]. Execution of PA-processes in turn is described by a structured operational semantics (SOS) [29]. In this way they could apply results about model-checking of PA-processes to flow analysis. For our purposes the approach chosen here is sufficient and produces less notational overhead.

Based on the operational semantics we have defined some run sets of interest and developed constraint systems that characterize these run sets. The constraint systems for same-level runs and reaching runs are essentially the ones used by Seidl and Steffen [33]. Also the constraint systems for inverse same-level runs and terminating runs are indicated in their work. The constraint system for bridging runs, however, is completely new. A further difference is that Seidl and Steffen *postulate* their constraint systems, while we use an operational semantics as a reference point. While this might be considered a minor or even trivial difference, in our opinion an operational justification of the constraint systems largely increases our understanding of what exactly is specified by the constraint systems.

Many reasonable variants of the run sets in question may be considered. For example, one could define reaching runs by

$$\mathsf{R}'(u) = \{\hat{r} \mid e_{Main} \stackrel{r}{\Longrightarrow} c \Longrightarrow \varepsilon, At_u(c)\}\,.$$

This definition deviates from the standard definition in that it considers only configurations c from which termination is possible, i.e., it characterizes the runs that both reach u and can be completed to a terminating run. In general, if assumption ASS2 is violated, this definition gives rise to smaller run sets than the standard definition. It might be preferable, if one is interested in terminating runs of programs only, like in total correctness reasoning. Similarly, many reasonable variants of the other run sets are conceivable and by techniques similar to the ones of Section 2.5.7 sound and complete constraint systems for these variants can be constructed. Operational specifications of the run sets in question allows to distingush these variants much more clearly than implicit specifications by means of constraint systems.

Validating constraint systems with respect to an operational semantics has another advantage: it helps to uncover subtle bugs. In the absence of an operational semantics Seidl and Steffen, for instance, fail to notice that constraint [R3] in the constraint system for reaching runs is not rich enough to characterize all reaching runs in a parallel composition if assumption ASS2 does not hold. We detected this error while trying to argue carefully the soundness of the constraint system. As a consequence their constraint system for reaching runs is unsound in the general case. To be fair, we should note that this does not affect the soundness of their analysis procedure that is not directly based on the constraint system for reaching runs. We should also say that they solve the problems that arise when assumption ASS1 is violated correctly. Here they validly propose to remove edges leaving unreachable program points before the analysis. This has essentially the same effect as the side conditions of the form $\mathsf{S}(u) \neq \emptyset$ added to the various constraints in Section 2.5.7.

Chapter 3

Non-Atomic Execution

The idealization that assignments execute atomically is quite common in the literature on program verification as well as in the theoretical literature on flow analysis of parallel programs. However, in a multi-processor environment where a number of concurrently executing processors share a common memory this assumption is hardly realistic. In such an environment two threads of control may well interfere while each of them is in the process of executing an assignment. The reason is that assignments are broken into smaller instructions before execution.

As a simple example, consider a program consisting of two parallel assignments both incrementing a variable x:

$$x := x + 1 \parallel x := x + 1\,.$$

Let us assume that x holds 0 initially. If assignments execute atomically, this program clearly will increment x twice and so terminate in a state in which variable x holds 2. However, in a multi-processor environment this program may well set x to 1. For example, the following execution may happen: first, one of the processors accesses the memory in order to get the value of x. While it is in the process of incrementing this value, but before it has written back the result, the second processors may access the memory, too, in order to get the value of x. In such a run, both processors read the initial value 0 for x, both will increment just this value, and both will write back 1 for x. Consequently, the program will terminate in a state where x holds 1 instead of 2.

In order to be more specific and, at the same time, keep the discussion simple, let us assume that the processors are stack machines. Then a compiler might generate the following piece of code for the assignment $x := x + 1$:

```
1   PUSH x
2   PUSH 1
3   ADD
4   POP x
```

Using unprimed numbers for the statements of the first processor and primed numbers for the statements of the second one, the two processors may then, e.g., execute their instructions in the following order:

$$1, 2, 1', 2', 3', 4', 3, 4\,.$$

We leave it to the reader to check that this execution indeed increments x just by 1.

The morale of this discussion is that, in the 'real world' of multi-processor execution, we cannot assume atomic execution of assignments. What we typically *may* safely assume, however, is that single reads of variables and single writes of variables are atomic, because the access to the memory is usually synchronized, e.g., through a common bus.

This said, we should mention that there are indeed execution scenarios for concurrent programs that guarantee atomic execution of assignments. In particular in a time-shared multi-tasking environment, where concurrent execution of threads is simulated by a single processor that switches between execution of code pieces implementing the different threads, assuming atomic execution of assignments may be safe, if context switches happen only between assignments, but not in the process of executing the code implementing a single assignment. The built-in scheduler of the Transputer, for instance, performs context switches only after certain types of instructions that typically end execution of assignment code [17].[1]

Note how non-atomic execution of assignments was modeled in the above example: first each assignment was broken into the smaller instructions of the stack machine; each of these instructions may be considered as an atomic unit of execution. Then the two threads $1, 2, 3, 4$ and $1', 2', 3', 4'$ of more fine-grained stack machine instructions was interleaved. This example tells us that we can develop an interleaving semantics for parallel programs that adequately models non-atomic execution of assignments by means of breaking assignments into more fine-grained atomic actions, an observation that is exploited in a moment.

The purpose of this section is to provide parallel flow graphs with an interleaving semantics that models non-atomic execution of assignments adequately. For this purpose we define a domain NR of sets of (non-atomic) runs and provide adequate definitions for the constants and operators used in the constraint systems in Section 2.5. Specifically, we provide

- an interpretation $[\![e]\!] \in \mathsf{NR}$ for the non-atomic runs of a base edge; and

[1] The Transputer designers have choosen this strategy in order to make context switches cheap and fast. In typical code, the contents of certain registers used for expression evaluation is no longer needed after such instructions. Therefore, these registers are not stored during context switches, which makes context switches fast. Actually, it is the compiler writer's task to ensure that the generated code does not rely on the registers keeping their contents after such instructions. Atomic execution of assignments in typical code is a neat side-effect of this design.

- interpretations for the operators ;, $\otimes$, *pre*, and *post* used in the constraint systems.

Solving the constraint systems from Section 2.5 over this new interpretation immediately gives us adequate definitions for the reaching, terminating, and u-v-runs of a parallel flow graph when assignments execute non-atomically.

3.1 Modeling Non-Atomic Execution by Virtual Variables

Suppose given a parallel flow graph and let X be the set of *program variables* which the statements of the flow graph refer to. In order to explain the meaning of non-atomic statements appropriately suppose furthermore given an infinite set V of *virtual (or internal) variables* disjoint from X. Intuitively, virtual variables are used to store intermediate results that are private to the threads. The parallel composition (or interleaving) operator defined later ensures that parallel threads do not interfere on virtual variables. We use the letters x, y to range over X, u, v to range over V, and the letters a, b to range over $X \cup V$.

For the purpose of the semantics, assignments are split into atomic operations. As an example consider an assignment statement $x := e(y_1, \ldots, y_k)$ in the program; $x, y_1, \ldots, y_k$ are program variables. There are many sensible atomicity assumptions. For example, we could work with the rather pessimistic assumption that just reads and writes of variables are atomic, then $x := e(y_1, \ldots, y_k)$ is replaced by a sequence of assignments

$$v_{\pi(1)} := y_{\pi(1)}\,; \ldots ; v_{\pi(k)} := y_{\pi(k)}\,; x := e(v_1, \ldots, v_k)\,,$$

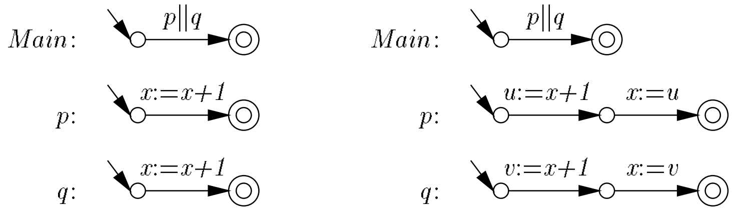
where $v_1, \ldots, v_k$ are arbitrary distinct virtual variables and π is a permutation of $\{1, \ldots, k\}$. The idea is that the other threads can execute atomic operations between these assignments.

More coarse-granular atomicity assumptions can be captured in a similar way. E.g. if we assume that the evaluation of the right-hand-side expressions is atomic then we would replace $x := e(y_1, \ldots, y_k)$ by

$$v := e(y_1, \ldots, y_k)\,; x := v\,.$$

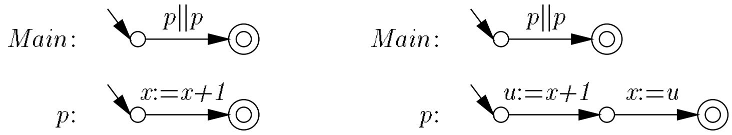
The important point to notice is that—whatever the specific atomicity assumption may be—if we assume that the execution of all assignments is non-atomic, then all assignments with a *program variable* on the left hand side have only *virtual variables* on the right hand side. Thus, all assignments belong to the set

$$\mathsf{Asg} \;=\; \{a := e(b_1, \ldots, b_k) \mid a \in X \;\Rightarrow\; b_1, \ldots, b_k \in L\}\,.$$



a) Original program. b) Transformed Program.

Figure 3.1: Introduction of virtual variables.



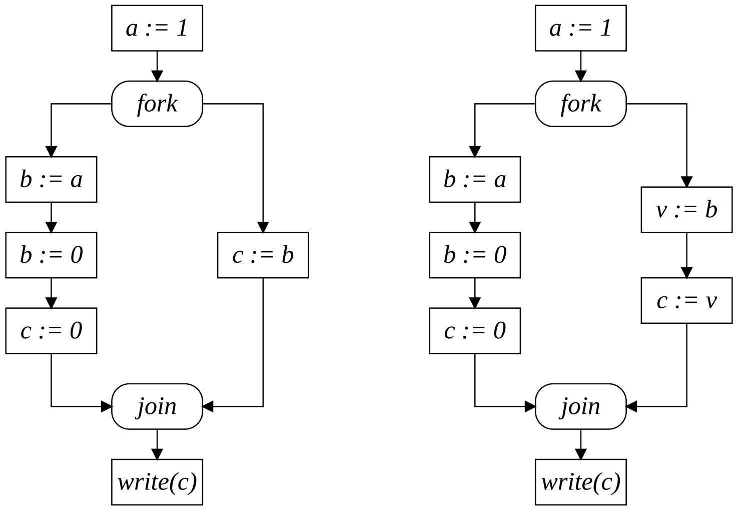
a) Original program. b) Transformed Program.

Figure 3.2: Confusion of virtual variables.

One way of obtaining a semantics for non-atomically executing assignments is to transform the assignments in the program prior to semantic interpretation. As an example consider the program in Fig. 3.1 a) which corresponds to the example discussed in the introduction. We could transform it to the program in Fig. 3.1 b) and then apply the standard interpretation.

The problem with this approach is that we must be careful not to confuse virtual variables of different threads. This is simple if only instances of different procedures run in parallel: then we can simply use different names for the virtual variables in different procedures. However, it becomes problematic if different instances of the same procedure may run in parallel like in the program in Fig. 3.2. Then we must model the virtual variables by local variables of the procedures which is not supported by the interpretation of flow graphs developed up to now. Therefore, we are using a different approach. We do not transform flow graphs but incorporate the transformation implicitly into the semantic interpretation of assignments.

Before we turn to the technical details of the new semantic interpretation we present an example that it makes a difference for constant detection whether base statements are assumed to execute atomically or not. This example illustrates that the main mechanism underlying the undecidability proof of interprocedural parallel constant detection [26] does not carry over to the non-atomic case.



(a) Original program (b) A virtual variable has been introduced

Figure 3.3: Introduction of a virtual variable.

3.2 A Motivating Example

Consider the following program for which a control flow graph-like representation is shown in Figure 3.3 (a):

$$a := 1; [(b := a; b := 0; c := 0) \parallel c := b] ; \mathbf{write}(c) .$$

Assume first that assignment statements execute atomically. It is not hard to show that under this assumption variable c is a (copy) constant of value 0 at the write instruction. In order to see this, note that in any execution $c := 0$ must be executed either after or before $c := b$ in the parallel thread. If it is executed after $c := b$ then c certainly holds 0 at the write statement because the value 0 is assigned to c in the last executed assignment, $c := 0$. On the other hand, if $c := 0$ is executed before $c := b$ then also the initialization of b, $b := 0$, must have been executed before $c := b$ such that $c := b$ also loads the value 0 to c.

The situation is dramatically different, if assignment statements may execute non-atomically. In particular, if the assignment $c := b$ in the second thread is executed non-atomically, the first thread may execute the two statements $b := 0$ and $c := 0$ that kill b and c after b is loaded from the common memory but before the loaded value is stored to c. This results in a run of the program that propagates the value 1 from the initialization $a := 1$ to the final write-statement.

As explained in the previous section, we may model the two stage non-atomic execution of $c := b$ by splitting it into two assignments $v := b$ and $c := v$, where v is a new *virtual variable* that cannot be accessed by the first thread (cf. Figure 3.3 (b)). We can think of virtual variable v as representing the register in which the value loaded from the common memory is stored. This register is private to the second thread (or rather the processor executing the second thread) and therefore there can be no interference on this variable. As there is no interference on virtual variable v, we can consider each of the virtual assignments $v := b$ and $c := v$ to be atomic. The resulting program has the run

$$r = \langle a := 1, b := a, v := b, b := 0, c := 0, c := v, \mathbf{write}(c)\rangle .$$

which—as the reader will have no difficulties to verify—propagates the value 1 from the initialization $a := 1$ to the write-statement. Thus, run r witnesses that c is not a copy constant at the write statement, in sharp contrast to the state of affairs under the assumption that assignments execute atomically.

3.3 The Domain of Non-Atomic Run Sets

A *(non-atomic) run* r is a sequence of assignments from the set Asg defined above: $\mathsf{Runs} = \mathsf{Asg}^*$. We write $\mathsf{virtual}(r)$ for the set of virtual variables appearing in run r and denote the empty run by ε and the concatenation operator by an infix dot or just by juxtaposition.

As the specific choice of virtual variables is immaterial, we assume that all considered sets of runs are closed under bounded renaming of virtual variables. This allows a simple and adequate definition of the composition operators. In order to allow a technically clean treatment of this assumption, let $\equiv \subseteq \mathsf{Runs} \times \mathsf{Runs}$ be the equality of runs up to bounded renaming of virtual variables, i.e. $r \equiv r'$ hold if and only if r' can be obtained from r by bounded renaming of virtual variables.

Proposition 3.1 *$\equiv$ is an equivalence.* □

For a set of runs $R \subseteq \mathsf{Runs}$ we write $R^{\equiv}$ for the closure of R w.r.t. $\equiv$:

$$R^{\equiv} = \{r \in \mathsf{Runs} \mid \exists r' \in R : r \equiv r'\} .$$

Obviously, this defines a closure operator.

Proposition 3.2

1. $R \subseteq R^{\equiv}$.

2. $(R^{\equiv})^{\equiv} = R^{\equiv}$.

3. $R \subseteq S$ *implies* $R^{\equiv} \subseteq S^{\equiv}$. □

The domain NR is given by the sets of runs that are closed under $\equiv$:

$$\mathsf{NR} = \{R \subseteq \mathsf{Runs} \mid R = R^{\equiv}\}.$$

The members of NR model sets of runs in a scenario where assignments execute non-atomically.

Lemma 3.3 $(\mathsf{NR}, \subseteq)$ *is a complete lattice with least element* $\bot_{\mathsf{NR}} = \emptyset$ *and greatest element* $\top_{\mathsf{NR}} = \mathsf{Runs}$.

Proof. $(\mathsf{NR}, \subseteq)$ is a sub-lattice of the power set lattice $(2^{\mathsf{Runs}}, \subseteq)$. To show this, we have to check, that NR is closed under arbitrary intersections and unions.

Here is the proof for intersection. Suppose $\mathcal{R} \subseteq \mathsf{NR}$ and $r, r' \in \mathsf{Runs}$ with $r \equiv r'$. We have to show that $r \in \bigcap \mathcal{R}$ if and only if $r' \in \bigcap \mathcal{R}$ which is simple:

$$
\begin{array}{ll}
 & r \in \bigcap \mathcal{R} \\
\text{iff} & [\text{Definition of } \bigcap \mathcal{R}] \\
 & \forall R \in \mathcal{R} : r \in R \\
\text{iff} & [\mathcal{R} \subseteq \mathsf{NR}, \text{ hence all } R \in \mathcal{R} \text{ are closed under } \equiv] \\
 & \forall R \in \mathcal{R} : r' \in R \\
\text{iff} & [\text{Definition of } \bigcap \mathcal{R}] \\
 & r' \in \bigcap \mathcal{R}.
\end{array}
$$

The proof for unions is just as simple and, therefore, omitted.

The least and greatest element of $(\mathsf{Runs}, \subseteq)$ are $\emptyset$ and Runs, respectively. It is obvious that both of them are closed under $\equiv$ and hence are also the least and greatest elements, respectively, of $(\mathsf{NR}, \subseteq)$. $\square$

In the sections that follow we provide definitions for the operators and constants appearing in the constraint systems and show their well-definedness.

3.3.1 Base Statements

We can work with various atomicity assumptions as discussed above. The most natural and conservative one is that just single reads and writes of variables are atomic. This is captured by defining the semantics of an assignment statement, $[\![x := e]\!] \in \mathsf{NR}$, where $y_1, \ldots, y_k$ are the variables appearing in e, as the set of runs of the form

$$\langle v_{\pi(1)} := y_{\pi(1)}, \ldots, v_{\pi(k)} := y_{\pi(k)}, x := e(v_1, \ldots, v_k)\rangle,$$

where π is a permutation of $\{1, \ldots, k\}$ and $v_1, \ldots, v_k$ are arbitrary distinct virtual variables. It is readily verified that $[\![x := e]\!]$ is well-defined, i.e., that $[\![x := e]\!] \in \mathsf{NR}$. We have to show that $[\![x := e]\!]$ is closed under $\equiv$ which is obvious as we admitted an arbitrary choice of virtual variables.

We may also work with a more coarse-grained semantics of assignments. For our purposes the choice is arbitrary, as the dependence trace abstraction of an assignment will be precise with respect to any of these definitions.

Obviously, the only non-atomic run of statement ***skip*** is the empty run. Hence, $[\![\textit{\textbf{skip}}]\!] = \{\varepsilon\}$. Obviously, $[\![\textit{\textbf{skip}}]\!] \in \mathsf{NR}$.

The non-atomic runs induced by a base edge $e \in \mathsf{Base}$ are the non-atomic runs of the statement associated with e: $[\![e]\!] = [\![A(e)]\!]$, where $A(e)$ is the base statement associated with base edge e in the underlying flow graph.

3.3.2 Sequential Composition

The *sequential composition operator*, $\cdot\,;\cdot : \mathsf{NR} \times \mathsf{NR} \to \mathsf{NR}$, which is written as an infix operator, is defined by

$$R\,;S = \{r \cdot s \mid r \in R, s \in S, \mathsf{virtual}(r) \cap \mathsf{virtual}(s) = \emptyset\}^{\equiv}\,.$$

Recall that $\cdot$ denotes concatenation of sequences. The condition about the local variables ensures that runs composed sequentially do not interact on local variables. It could be replaced by a condition that in a run all local variables are initialized before they are used. However, the latter condition would not be preserved by the pre-operator and, therefore, we prefer the chosen solution. The outer closure operator ensures that ; is well-defined

3.3.3 Interleaving Operator

In order to define the interleaving (or parallel composition) operator some notation is needed. Let $r = \langle e_1, \ldots, e_n\rangle$ be a sequence and $I = \{i_1, \ldots, i_k\}$ a subset of positions in r such that $1 \le i_1 < i_2 < \cdots < i_k \le n$. Then $r|I$ is the sequence $\langle e_{i_1}, \ldots, e_{i_k}\rangle$. We write $|r|$ for the length of r, viz. n. The *interleaving operator*, $\otimes : \mathsf{NR} \times \mathsf{NR} \to \mathsf{NR}$, which we write in an infix form, is defined by

$$\begin{aligned} R \otimes S \;\; = \;\; \{r \mid \exists I_R, I_S :\; & I_R \cup I_S = \{1, \ldots, |r|\}, I_R \cap I_S = \emptyset, \\ & r|I_R \in R, r|I_S \in S, \mathsf{virtual}(r|I_R) \cap \mathsf{virtual}(r|I_S) = \emptyset\}^{\equiv}\,. \end{aligned}$$

The condition about the local variables in $r|I_R$ and $r|I_S$ ensures that parallel threads do not exchange values via local variables. The application of the closure operator $(\cdot)^{\equiv}$ guarantees well-definedness: $R \otimes S \in \mathsf{NR}$ for $R, S \in \mathsf{NR}$.

Suppose $r, s, t \in \mathsf{Runs}$ with $\mathsf{virtual}(r) \cap \mathsf{virtual}(s) = \emptyset$. We call t an *interleaving of* r *and* s if

$$\exists I_r, I_s : I_r \cup I_s = \{1, \ldots, |r|\}, I_r \cap I_s = \emptyset, t|I_r = r, t|I_s = s$$

and denote the set of interleavings of r and s by $r \otimes s$.

3.3.4 Pre-Operator

The pre-operator, $pre : \mathsf{NR} \to \mathsf{NR}$ is defined as follows:

$$pre(R) \quad = \quad \{r \in \mathsf{Runs} \mid \exists r' \in \mathsf{Runs} : r \cdot r' \in R\} \, .$$

Lemma 3.4 *pre is well-defined.*

Proof. We have to show that, for any $R \in \mathsf{NR}$, $pre(R)$ is closed under $\equiv$. So suppose given $r, s \in \mathsf{Runs}$ with $s \equiv r \in pre(R)$. Then there is $r' \in \mathsf{Runs}$ with $r \cdot r' \in R$. By bounded renaming of local variables in r' we can construct a run s' such that $s \cdot s' \equiv r \cdot r'$. As R is closed under $\equiv$, $s \cdot s' \in R$ and hence $s \in pre(R)$.□

3.3.5 Post-Operator

Analogously to the pre-operator, the post operator $post : \mathsf{NR} \to \mathsf{NR}$ is defined as follows:

$$post(R) \quad = \quad \{r \in \mathsf{Runs} \mid \exists r' \in \mathsf{Runs} : r' \cdot r \in R\} \, .$$

Lemma 3.5 *post is well-defined.* □

3.4 Conclusion

We have defined a complete lattice $(\mathsf{NR}, \subseteq)$ the members of which model sets of runs in a scenario in which assignment statements execute non-atomically. In order to enable an interleaving semantics to adequately capture the effect of non-atomic execution of assignments, we resorted to *virtual variables* that model storage locations that are private to the processor that executes a thread.

The members of NR are those sets of runs that are bounded under renaming of virtual variables. We have provided definitions for the operators and constants appearing in the constraint systems that capture reaching, terminating, and u-v-runs in a parallel flow graph. The (smallest) solution of these constraint systems over this new interpretation induces a new semantics of parallel flow graphs that captures non-atomic execution of assignments. The new semantics provides a reference point for assessing flow analyses that are performed by means of an abstract interpretation of the constraint systems. We will put this idea to advantage in Chapter 4 where we show that the dependence trace interpretation is a precise abstraction of the non-atomic interpretation of parallel flow-graphs.

Chapter 4

Dependence Traces

We can indirectly detect copy constants and eliminate faint code on the basis of the following information: given a program point u and a variable x of interest; when control is at another program point v, which variables y may influence the value of x at u? This information is the abstraction of u-v-runs (bridging runs) to the set of *dependences* mediated by these runs. Here we mean by a dependence mediated by a run r a pair of variables (x, y) such that the value of y after execution of r depends on the initial value of x (cf. Section 4.1 for the formal definition).

In Section 2.5 a constraint system characterizing the set of bridging runs of a given parallel flow graph was given. We would like to perform the analysis by evaluating this constraint system over an abstract interpretation. Unfortunately, we cannot use dependences themselves as abstract domain because, in general, we cannot obtain the dependences of a parallel composition of run sets from the dependences of the components (cf. Section 4.2). Therefore, an abstraction employing just dependences cannot be sound and complete at the same time. We need to collect more information in the abstract domain.

In this chapter, we introduce an adequate abstract domain from which on the one hand dependences can be inferred easily and for which on the other hand abstract operation can be defined that mirror precisely the corresponding operations on sets of (non-atomic) runs.

The basic idea is to collect not only dependences but sequences of dependences (*dependence sequences*) that can successively be mediated by a run. For example, the run $r_1 = \langle c := b, e := d\rangle$ has $\langle (b, c), (d, e)\rangle$ as one of its dependence sequences. This dependence sequence plays a dual role: on the one hand, it captures the potential of r_1 to mediate the dependence (b, e) if a parallel run fills the 'gap' between c and d (like, e.g., $r_2 = \langle d := c\rangle$) and, on the other hand, its potential to successively fill the 'gaps' (b, c) and (d, e) in a parallel run (like, e.g., in $r_3 = \langle b := a, d := c, f := e\rangle$.

Further information must be collected. To see why, compare the run $r_4 = \langle b := 0, c := b, e := d, e := 0\rangle$ to r_1. Unlike r_1, r_4 does *not* have the potential to mediate the dependence (b, e) if a parallel run fills the 'gap' between c and d, but it is still able to successively fill the gaps (b, c) and (d, e) in a parallel run. The difference is that in r_4, unlike in r_1, the part of the run before b is read and after e is written is not transparent for b and e, respectively. Therefore, we refine dependence sequences to *dependence traces* in which we record in addition to a dependence trace by two Boolean values, whether the parts of the run before the source variable of the first dependence is read and after the target variable of the final dependence is written are transparent for these variables. Run r_1 for instance has the dependence trace $(1, \langle (b, c), (d, e) \rangle, 1)$ which r_4 has not, but both share the dependence trace $(0, \langle (b, c), (d, e) \rangle, 0)$. In order to allow a proper propagation of transparency information in sequential composition, we furthermore collect the set of variables for which a transparent run exists in the abstract domain in addition to the dependence traces.

According to these ideas, we can abstract a set of (non-atomic) runs R to a pair (T_R, D_R) consisting of the set of variables

$$T_R = \{x \mid \exists r \in R : r \text{ is transparent for } x\}$$

and the set of dependence traces

$$D_R = \{\tau \mid \exists r \in R : \tau \text{ is a dependence trace of } r\}\,.$$

On this abstraction of run sets, we can indeed define abstract operators that precisely mirror the operators on sets of non-atomic runs that are used in the constraint systems of Section 2.5. However, we are not yet done. The problem is, that this abstract domain is not effective, because D_R can be infinite. In order to obtain an effective domain, we have to go one step further.

For this purpose, we define a subsumption order, written $\sqsubseteq$, on transition traces. The intuition is that a transition trace τ is subsumed by another transition trace τ' if τ' has fewer gaps than τ—we write this as $\tau \sqsubseteq \tau'$ (cf. Section 4.4). Intuitively, τ' is more useful than τ in forming dependences. We then collect for a run set only the transition traces that are maximal with respect to the order $\sqsubseteq$. This set forms an antichain with respect to $\sqsubseteq$. It is not hard to show that all $\sqsubseteq$-maximal dependence traces of a run set are *short* in a certain sense made precise in Section 4.6. As there are only finitely many short dependence traces this makes the abstract domain finite, such that we can effectively perform fixpoint calculations.

Summarizing, the abstract domain consists of pairs (D, T) where D is an $\sqsubseteq$-antichain of short dependence traces and T is a set of variables. It is not hard to define on this domain abstract counterparts to the sequential composition operator and to the pre- and post-operator on run sets and to show that these abstract operators are precise abstractions of the concrete ones. It is also straightforward to abstract the run sets associated with base edges precisely.

The interleaving operator, however, poses some complication. The natural way to compose two transition traces τ and τ' concurrently is to use τ to fill gaps in τ' and vice versa. This was our motivation for considering transition traces in the first place; a precise formalization of this idea is given through the relation C in Section 4.11.1. However, if τ'' is a transition trace obtained in this way from a transition trace τ of a run r and a transition trace τ' of a run r', it is not obvious that there is always a run constructed by interleaving r and r' that has τ'' as one of its transition traces. Otherwise the abstraction would be imprecise. Indeed this would fail for run sets deriving from an atomic interpretation of base statements but we can show this for run sets deriving from a non-atomic interpretation of base statements.

On the other hand, it is quite typical that rather short transition traces are obtained from considerably longer ones in this way. There is thus some reason to suspect that we cannot obtain all $\sqsubseteq$-maximal dependence traces of the interleaving $R_1 \otimes R_2$ of two run sets from the $\sqsubseteq$-maximal dependence traces of the components. This would render the abstract operator unsound. Fortunately, we can show that this is not the case. The main insight is covered by a shortening lemma, Lemma 4.38.

As an auxiliary notion we introduce a further order on dependence traces, called the *implication order* which is written as $\leq$. Its name is justified by the fact that any run r that has τ as a dependence trace also has τ' as a dependence trace, if $\tau \leq \tau'$. Therefore, the implication order captures implied knowledge about dependence traces of runs, hence its name. The implication order is crucial in particular for a concise formulation of the shortening lemma mentioned above.

In the remainder of this chapter we elaborate these topics in detail.

4.1 Transparency and Dependences

A run r is called *transparent* for a variables a if it does not contain an assignment with a as left hand side variable. Thus, a run is transparent for a if its execution is guaranteed not to change the value held by a.

Example 4.1 *The run $\langle a := 0, b := c\rangle$ is not transparent for variables a and b but for all other variables, in particular for c.* □

A *dependence* is a pair $d = (x, y)$ of program variables $x, y \in X$. We call x the *source variable* and y the *destination variable* of d. A run r is said to *mediate the dependence* (x, y), if there are variables $a_0, \ldots, a_l$, $l > 0$, expressions $e_1, \ldots, e_l$, and (sub-) runs $r_0, \ldots, r_l$ such that

1. $r = r_0 \cdot \langle a_1 := e_1\rangle \cdot r_1 \cdot \langle a_2 := e_2\rangle \cdot r_2 \cdot \ldots \cdot \langle a_l := e_l\rangle \cdot r_l$;

2. $a_0 = x$, $a_l = y$;

3. e_i contains a_{i-1} for $i = 1, \ldots, l$; and

4. r_i is transparent for a_i for $i = 0, \ldots, l$.

Example 4.2 *The run* $\langle b := 0, b := a, c := b, e := 0, f := e\rangle$ *mediates the dependences* (a, b), (a, c), *and* (b, c) *but not the dependence* (e, f) *because* e *is killed by the assignment* $e := 0$ *before it is read.* □

4.2 Dependence Traces

In general, the dependences of the interleaving $R_1 \otimes R_2$ of two sets of runs cannot directly be inferred from the dependences of the component run sets R_1 and R_2. As an example, consider the two sets of runs $R_1 := \{\langle b := a, d := c\rangle\}$ and $R_2 := \{\langle b := a\rangle, \langle d := c\rangle\}$. Both mediate just the dependences (a, b) and (c, d). But the interleaving of R_1 with $R_3 := \{\langle c := b\rangle\}$ contains the run $\langle b := a, c := b, d := c\rangle$ that mediates the dependence (a, d) while there is no run in the interleaving of R_2 and R_3 that mediates this dependence.

Thus, an abstraction of sets of runs that faithfully mirrors dependences must collect more information than just dependences. We propose to employ *dependence traces* that are defined in the remainder of this section.

The basic idea is to collect not just dependences but sequences of dependences that can successively be mediated by a run. For example, we would record the sequences $\varphi = \langle (a, b), (c, d)\rangle$ for the run $r_1 = \langle b := a, d := c\rangle$ from R_1 but not for R_2. Intuitively, φ shows us that r_1 could mediate a dependence from a to d if a parallel component fills the gap from b to c. Dually, it also indicates that r_1 can successively fill the gaps (a, b) and (c, d).

A *dependence sequence* is a sequence $\varphi = \langle (x_1, y_1), \ldots, (x_k, y_k)\rangle$, $k \geq 0$, of dependences. We also allow the empty dependence sequence ε. This mostly smoothens the exposition that follows but sometimes requires a special treatment. We write $\overleftarrow{\varphi}$ for x_1 and $\overrightarrow{\varphi}$ for y_k, if $\varphi \neq \varepsilon$; if $\varphi = \varepsilon$, $\overleftarrow{\varphi}$ and $\overrightarrow{\varphi}$ are undefined. We denote the set of transfer sequences by DS.

Example 4.3 $\varphi = \langle (a, b), (x, y)\rangle$ *is a dependence sequence with* $\overleftarrow{\varphi} = a$ *and* $\overrightarrow{\varphi} = y$. □

As explained in the introduction to this chapter, we must distinguish between runs like r_1 above and runs like $r_1' := \langle a := 0, b := a, d := c, d := 0\rangle$ by means of initial and final transparency bits. Unlike r_1, r_1' does not have the potential to mediate the dependence (a, d) if the gap (b, c) is filled by a parallel run but like r_1 it can successively fill the gaps (a, b) and (c, d).

A *dependence trace* is a triple $\tau = (\iota, \varphi, \kappa)$ consisting of Boolean values $\iota, \kappa \in \mathbb{B}$ coding initial and final transparency and a dependence sequence φ. We assume that $\iota = 0$ and $\kappa = 0$ if $\varphi = \varepsilon$. The set of dependence traces is denoted by DT:

$$\mathsf{DT} = \{(\iota, \varphi, \kappa) \in \mathbb{B} \times \mathsf{DS} \times \mathbb{B} \mid \varphi = \varepsilon \Rightarrow (\iota = 0 \wedge \kappa = 0)\} .$$

The dependence trace $\tau = (\iota, ((x_1, y_1), \ldots, (x_k, y_k)), \kappa)$ is called *compatible* with a run r, $r \vdash \tau$ for short, if there are sub-runs $t_0, \ldots, t_k, r_1, \ldots, r_k$, such that

1. $r = t_0 r_1 t_1 r_2 \cdots r_k t_k$;
2. r_i mediates the dependence (x_i, y_i) for $i = 1, \ldots, k$;
3. $\iota = 1$ implies that t_0 is transparent for x_1; and
4. $\kappa = 1$ implies that t_k is transparent for y_k.

In this case, we call $t_0 r_1 t_1 r_2 \cdots r_k t_k$ a decomposition of r that witnesses $r \vdash \tau$. Note that $r \vdash (0, \varepsilon, 0)$ holds for all runs r as witnessed by the trivial decomposition $r = t_0$. The trivial dependence trace $(0, \varepsilon, 0)$ allows us to distinguish the dependence trace abstraction of an empty run set from the abstraction of a run sets without interesting dependence traces.

Instead of saying "τ is compatible with r" we often use the phrase "τ is a dependence trace of r".

Example 4.4 *Consider the run $r = \langle a := 0, b := a, c := b, c := 0, f := e, e := 0\rangle$. One of the dependence traces of r is $\tau = (0, \langle (a, c), (e, f)\rangle, 1)$ as witnessed by the decomposition $r = t_0 r_1 t_1 r_2 t_2$ where*

$$\underbrace{a := 0}_{t_0}, \underbrace{b := a, c := b}_{r_1}, \underbrace{c := 0}_{t_1}, \underbrace{f := e}_{r_2}, \underbrace{e := 0}_{t_2} .$$

Another decomposition witnessing τ is

$$\underbrace{a := 0}_{t_0}, \underbrace{b := a, c := b}_{r_1} \underbrace{\ ,\ }_{t_1 = \varepsilon} \underbrace{c := 0, f := e, e := 0}_{r_2} \underbrace{\quad}_{t_2 = \varepsilon} .$$

The run r has also many other dependence traces, e.g., $(1, \langle (b, c), (e, f)\rangle, 1)$ and $(1, \langle (e, f)\rangle, 1)$. □

Ultimately, we are interested in dependence traces without gaps that code complete transfers from one variable to another one, where a gap can either be a lack of initial or final transparency or a hole from y_i to x_{i+1}. The other dependence traces are needed only to compute these 'perfect' dependence traces in a compositional fashion. Thus, the dependence traces of ultimate interest are those of the form $(1, (x, y), 1)$. They correspond to dependences.

Proposition 4.5 $r \vdash (1, (x, y), 1)$ *if and only if r mediates the dependence (x, y).*
□

We can abstract a set R of runs to the set $D_R := \{\tau \mid \exists r \in R : r \vdash \tau\}$ of compatible dependence traces and it is possible to define precise abstract operators on this abstraction.[1] However, this abstraction is not effective, because D_R is in general infinite.

Fortunately, it is not necessary to collect *all* compatible dependence traces in the abstraction, in order to describe the potential for forming dependences with a parallel context. It suffices to retain only certain 'short' dependence traces in the abstraction that subsume the potential of all the other ones. A number of definitions and observations are necessary to make this precise. However, before we turn to the technical development, let us illustrate this kind of subsumption by a small example.

Consider the two dependence traces $\tau_1 = (1, (a, b) \cdot (c, d) \cdot (e, f), 1)$ and $\tau_2 = (1, (a, d) \cdot (e, f), 1)$. Intuitively, both have the gap (d, e) but τ_1 has the additional gap (b, c). If a run r of a parallel context can successively fill the two holes in τ_1—i.e. if r is compatible with the dependence trace $\tau_3 = (0, (b, c) \cdot (d, e), 0)$—it can also fill the single hole in τ_2—i.e. r is then also compatible with $\tau_4 = (0, (d, e), 0)$. Two interesting relationships between dependence traces popped up in this discussion. On the one hand, τ is "subsumed" by τ' in the sense sketched above as it has fewer gaps. On the other hand τ_4 is "implied" by τ_3 as it has less dependences: any run having τ_3 as a dependence traces also has τ_4 as a dependence trace.

We now define two orders on the set of dependence traces that capture these two relationships, the "implication order" and the "subsumption order".

4.3 Implication Order

Let $\leq \subseteq \mathsf{DT} \times \mathsf{DT}$ be the smallest reflexive and transitive relation on the set of dependence traces that satisfies

1. $(\iota, \varphi \cdot (x, y) \cdot \psi, \kappa) \leq (\iota, \varphi \cdot \psi, \kappa)$, if $\varphi \neq \varepsilon \vee \iota = 0$ and $\psi \neq \varepsilon \vee \kappa = 0)$;
2. $(1, \varphi, \kappa) \leq (0, \varphi, \kappa)$; and
3. $(\iota, \varphi, 1) \leq (\iota, \varphi, 0)$.

Proposition 4.6 *$\leq$ is a partial order on* DT*.* □

The implication order $\leq$ allows us to weaken the information in a dependence trace in two ways. First of all, we can omit dependences (1.); here we must be careful not to omit the first or last dependence if the corresponding transparency bit is set, as otherwise the transparency bit might become invalid. Secondly, we

[1] For sequential composition we also need the set of variables for which a transparent run exists in order to allow a proper propagation of the transparency bits.

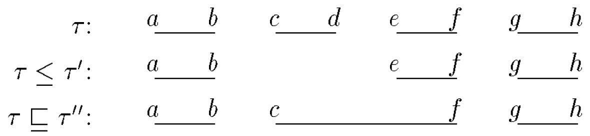


Figure 4.1: Illustration of implication and subsumption order.

can weaken the information about the transparency of the inital or final part of the run, by changing the transparency bits from 1 to 0 (2. & 3.).

The most appealing fact about $\leq$ is that it preserves compatibility, which justifies the name "implication order".

Proposition 4.7 ($\leq$ preserves compatibility) *Suppose $r \vdash \tau$ and $\tau \leq \tau'$. Then $r \vdash \tau'$.* □

Example 4.8 *Consider the dependence trace $\tau = (1, \langle (a,b), (c,d) \rangle, 0)$, which is, for instance, compatible with the run $r = \langle b := a, c := 0, d := c, d := 0 \rangle$. Here is a list of the dependence traces that are implied by τ:*

$$\begin{aligned} \tau_1 &= (0, \langle (a,b), (c,d) \rangle, 0) \\ \tau_2 &= (1, \langle (a,b) \rangle, 0) \\ \tau_3 &= (0, \langle (a,b) \rangle, 0) \\ \tau_4 &= (0, \langle (c,d) \rangle, 0) \\ \tau_5 &= (0, \varepsilon, 0) \end{aligned}$$

i.e., we have $\tau \leq \tau_i$ for $i = 1, \ldots, 5$. All of them are dependence traces of r. But we do not have $\tau \leq \tau_6$ for $\tau_6 = (1, \langle (c,d) \rangle, 0)$. And indeed, τ_6 is not a dependence trace of r because variable c is killed before it is read in r. □

4.4 Subsumption Order

A transfer information with fewer gaps is more useful for the construction of dependences. We now define an order $\sqsubseteq \subseteq \mathsf{DT} \times \mathsf{DT}$. Intuitively, $\tau \sqsubseteq \tau'$ captures that τ' has fewer gaps than τ and thus subsumes the potential of τ for forming dependences with a cooperating parallel context, hence we call $\sqsubseteq$ the *subsumption order*. We define $\sqsubseteq$ as the smallest transitive and reflexive relation that satisfies

$$(\iota, \varphi \cdot (x,y) \cdot \varphi' \cdot (x',y') \cdot \varphi'', \kappa) \sqsubseteq (\iota, \varphi \cdot (x,y') \cdot \varphi'', \kappa)\,.$$

In Fig. 4.1 we illustrate the difference between the implication and the subsumption order. For simplicity, we only show the dependence sequences and ignore the transparency bits. In the top row we show a dependence trace τ, in the middle row a dependence trace τ' that is implied by τ, and in the bottom row a dependence trace τ'' that subsumes τ. With the implication order, dependences can be omitted (and transparency bits weakened). In contrast with the subsumption order, gaps are removed.

It is obvious from the defining rule that a transfer sequence τ' that properly subsumes another transfer sequence τ embodies a strictly shorter transfer sequence. Therefore, $\sqsubseteq$ satisfies the ascending chain condition.

Proposition 4.9 *$\sqsubseteq$ is a partial order on* DT *that satisfies the 'ascending chain condition': every strictly increasing sequence $\tau_1 \sqsubset \tau_2 \sqsubset \cdots$ is finite.* □

Note that the dependence traces of the form $(1, (x, y), 1)$ (which by Proposition 4.5 correspond to dependences) are maximal w.r.t. $\sqsubseteq$. This simple observation is important, as it implies that we cover all dependences even when we only consider $\sqsubseteq$-maximal dependence traces.

4.5 A Lattice of Antichains

An *antichain* with respect to $\sqsubseteq$ (or $\sqsubseteq$-antichain for short) is a set $D \subseteq \mathsf{DT}$ of dependence traces satisfying

$$\neg\, \exists \tau, \tau' \in D : \tau \sqsubseteq \tau' .$$

We denote the set of $\sqsubseteq$-antichains by AC. We can lift the subsumption order to AC as follows:

$$D \sqsubseteq D' \qquad :\Leftrightarrow \quad \forall \tau \in D \exists \tau' \in D' : \tau \sqsubseteq \tau' .$$

The intuition is that D' subsumes D, if every single dependence trace in D is subsumed by a dependence trace in D'. We call $\sqsubseteq$ the antichain order. This is justified by the following lemma.

Lemma 4.10 *$\sqsubseteq$ is a partial order on* AC*.*

Proof. It is straightforward to show that $\sqsubseteq$ is reflexive and transitive. Let us show that $\sqsubseteq$ is also antisymmetric and hence a partial order:

Suppose $D \sqsubseteq D' \sqsubseteq D$. We show that $D \subseteq D'$, the reverse inclusion follows analogously. Suppose $\tau \in D$. Then there is $\tau' \in D'$ with $\tau \sqsubseteq \tau'$ as $D \sqsubseteq D'$. Because of $D' \sqsubseteq D$, there is $\tau'' \in D$ with $\tau' \sqsubseteq \tau''$. Thus, we have

$$D \ni \tau \sqsubseteq \tau' \sqsubseteq \tau'' \in D .$$

As D is an antichain, this implies that $\tau = \tau''$. Consequently, all these three dependence traces must be equal: $\tau = \tau' = \tau''$. But then $\tau = \tau' \in D'$. □

A simple way to form an $\sqsubseteq$-antichain out of an arbitrary subset $D \subseteq \mathsf{DT}$ is to consider the set of $\sqsubseteq$-maximal elements in D. We denote this set by $D^{\uparrow}$:

$$D^{\uparrow} = \{\tau \in D \mid \neg\exists \tau' \in D : \tau \sqsubset \tau'\}\,.$$

The dependence traces in $D^{\uparrow}$ subsume all dependence traces in D. In this sense, no interesting information is lost when going from D to $D^{\uparrow}$.

Lemma 4.11 (**$^{\uparrow}$ subsumes**) *For any $\tau \in D$ there is a $\tau' \in D^{\uparrow}$ such that $\tau \sqsubseteq \tau'$.*

The lemma follows from the ascending chain condition.

The operator $^{\uparrow}$ is a co-closure operator that yields $\sqsubseteq$-antichains:

Lemma 4.12 (**$^{\uparrow}$ is a co-closure operator**)

1. *$D^{\uparrow} \subseteq D$.*
2. *$(D^{\uparrow})^{\uparrow} = D^{\uparrow}$.*
3. *$D^{\uparrow}$ is an $\sqsubseteq$-antichain.*
4. *$(\cdot)^{\uparrow}$ is monotonic: $D \subseteq E$ implies $D^{\uparrow} \sqsubseteq E^{\uparrow}$.*

The proof of these properties is straightforward.

By 3., $(\cdot)^{\uparrow}$ is an operator from 2^{DT} to AC, by 4., this operator is monotonic. Indeed, as we will see in a minute, $(\cdot)^{\uparrow}$ is the lower adjoint of a Galois connection between 2^{DT} and AC. Before we elaborate this, let us show that the $\sqsubseteq$-antichains together with the lifted subsumption order form a complete lattice.

Lemma 4.13 $(\mathsf{AC}, \sqsubseteq)$ *is a complete lattice. The least upper bound (lub) of a subset $\mathcal{D} \subseteq \mathsf{AC}$ is $\bigsqcup \mathcal{D} := (\bigcup \mathcal{D})^{\uparrow}$ and the least element of* $(\mathsf{AC}, \sqsubseteq)$ *is* $\bot_{\mathsf{AC}} := \emptyset$.

Proof. In order to show that $(\mathsf{AC}, \sqsubseteq)$ is a complete lattice, it suffices to demonstrate that any subset $\mathcal{D} \subseteq \mathsf{AC}$ has a least upper bound. We show that, as claimed in the lemma, $E := (\bigcup \mathcal{D})^{\uparrow}$ is indeed the least upper bound of $\mathcal{D}$.

Firstly, E is an upper bound of $\mathcal{D}$: We have to show that $D \sqsubseteq E$ for any $D \in \mathcal{D}$, which is seen as follows:

$$\begin{array}{ll} & \tau \in D \\ \Rightarrow & [D \in \mathcal{D}, \text{definition of } \bigcup \mathcal{D}] \\ & \tau \in \bigcup \mathcal{D} \\ \Rightarrow & [\text{Lemma 4.11, definition } E] \\ & \exists \tau' \in E : \tau \sqsubseteq \tau'\,. \end{array}$$

Secondly, E is smaller than any other bound $\mathcal{D}$: Suppose F is an arbitrary upper bound of $\mathcal{D}$. Then $E \sqsubseteq F$ follows from the following chain of implications:

$$
\begin{array}{ll}
 & \tau \in E \\
\Rightarrow & \text{[Definition } E\text{, Lemma 4.12(1.)]} \\
 & \tau \in \bigcup \mathcal{D} \\
\Rightarrow & \text{[Definition of } \bigcup \mathcal{D}\text{]} \\
 & \exists D \in \mathcal{D} : \tau \in D \\
\Rightarrow & [D \sqsubseteq F \text{ as } F \text{ is an upper bound of } \mathcal{D}\text{, definition } \sqsubseteq] \\
 & \exists \tau' \in F : \tau \sqsubseteq \tau' .
\end{array}
$$

The least element of $(\mathsf{AC}, \sqsubseteq)$ is $\bot_{\mathsf{AC}} = \bigsqcup \emptyset = (\emptyset)^{\uparrow} = \emptyset$. □

Let us consider another operator on sets of dependence traces, the downwards closure operator $(\cdot)^{\downarrow}$: for a set $D \in \mathsf{DT}$ it is defined by

$$D^{\downarrow} = \{\tau \in \mathsf{DT} \mid \exists \tau' \in D : \tau \sqsubseteq \tau'\} .$$

We can apply $(\cdot)^{\downarrow}$ in particular to antichains. Thus, we may consider $(\cdot)^{\downarrow}$ as an operator $(\cdot)^{\downarrow} : \mathsf{AC} \to 2^{\mathsf{DT}}$. It is not hard to see that $(\cdot)^{\downarrow}$ is monotonic.

Proposition 4.14 *Suppose $A, B \in \mathsf{AC}$. Then $A \sqsubseteq B$ implies $A^{\downarrow} \subseteq B^{\downarrow}$.* □

$(\cdot)^{\uparrow}$ and $(\cdot)^{\downarrow}$ are approximate inverses of each other.

Lemma 4.15 *For any $D \in \mathsf{DT}$, we have $D^{\uparrow\downarrow} \supseteq D$ and $D^{\downarrow\uparrow} \subseteq D$. For any $A \in \mathsf{AC}$, we even have $A^{\downarrow\uparrow} = A$. As a consequence, $((\cdot)^{\uparrow}, (\cdot)^{\downarrow})$ is a Galois surjection from 2^{DT} to AC:*

$$2^{\mathsf{DT}} \underset{(\cdot)^{\downarrow}}{\overset{(\cdot)^{\uparrow}}{\rightleftarrows}} \mathsf{AC}$$

Proof.

$D^{\uparrow\downarrow} \supseteq D$: By Lemma 4.11, there is, for any $\tau \in D$, a dependence trace $\tau' \in D^{\uparrow}$ such that $\tau \sqsubseteq \tau'$. This implies that $\tau \in D^{\uparrow\downarrow}$.

$D^{\downarrow\uparrow} \subseteq D$: If $\tau \in D^{\downarrow\uparrow}$, then τ is a maximal element in $D^{\downarrow}$. The maximal elements in $D^{\downarrow}$, however, must already be in D, as they cannot be added to D by lying strictly below another element of D.

$A^{\downarrow\uparrow} = A$: It remains to show that $A^{\downarrow\uparrow} \supseteq A$. Any $\tau \in A$ is maximal in $A^{\downarrow}$ because A is an antichain. Therefore, any such τ is also in $A^{\downarrow\uparrow}$. □

The fact that $((\cdot)^{\uparrow}, (\cdot)^{\downarrow})$ is a Galois surjection from 2^{DT} into AC shows us that $\sqsubseteq$-antichains form a reasonable abstraction of sets of dependence traces. It also has other interesting consequences.

First of all, it implies that $(\cdot)^{\uparrow}$ is universally disjunctive, which is important for ensuring that the abstraction mapping and the abstract operators defined later are universally disjunctive as well.

Proposition 4.16 $(\cdot)^{\uparrow} : 2^{\mathsf{DT}} \to \mathsf{AC}$ *is universally disjunctive ('distributive').* $\square$

Secondly, it shows us that we can present $(\mathsf{AC}, \sqsubseteq)$ isomorphically by downwards closed sets of dependence traces. From the theory of Galois connections, we know that the images of the upper and lower adjoint are isomorphic. This implies that $(\mathsf{AC}, \sqsubseteq)$, the image of $(\cdot)^{\uparrow}$, is isomorphic to the image of $(\cdot)^{\downarrow}$, which is the set of downwards closed sets of dependence traces ordered by set inclusion. Note that this isomorphism depends on the fact that the underlying subsumption order on dependence traces satisfies the ascending chain condition. Otherwise, Lemma 4.11 would fail and we would not have the property $D^{\uparrow\downarrow} \supseteq D$ that is crucial for the isomorphism between antichains and downwards closed sets.

For our purposes it is more convenient to work with antichains, because this leads to a more natural definition of the interleaving operator. If we work with downwards closed sets we add dependence traces by means of downwards closure that are not compatible with any run in the abstracted run set. These additional dependence traces do not represent actual potential of the run set and in order to avoid imprecision, we must ensure that they are not considered for inferring dependence traces of interleavings. Therefore, we can only use the $\sqsubseteq$-maximal dependence traces in a downwards closed set, because only they can be identify as actual dependence traces of the abstracted run set. But then we can just as well collect just these maximal dependence traces which amounts to working with $\sqsubseteq$-antichains.

4.6 Short Dependence Traces

A dependence sequence $\varphi = \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle$ is called *short* if

1. all destination variables of dependences not counting the last one are distinct: for all $1 \le i < j < k$, $y_i \ne y_j$; and

2. all source variables of dependences not counting the first one are distinct: for all $1 < i < j \le k$, $x_i \ne x_j$.

A dependence trace $\tau = (\iota, \varphi, \kappa)$ is called *short* if the embodied dependence sequence φ is short. We write DTS for the set of short dependence traces:

$$\mathsf{DTS} \quad = \quad \{\tau \in \mathsf{DT} \mid \tau \text{ is short}\}\,.$$

Example 4.17 *Consider the run $r = \langle c := a, c := b, e := d \rangle$. One of its dependence traces is $\tau = (1, \langle (a,c),(b,c),(d,e) \rangle, 1)$, which is not short due to the repetition of variable c as a target variable. But run r has also the dependence trace $\tau' = (1, \langle (a,c),(d,e) \rangle, 1)$ which is short and subsumes τ. This is not a coincidence as we will see in a moment (Lemma 4.19).* □

We are interested in short dependence traces for two reasons. Firstly, there are only finitely many short dependence traces. This makes the abstract domain introduced in the next section finite as well and ensures that fixpoints for monotonic functions on this domain can be calculated effectively. The following lemma provides a formula for the cardinality of DTS and an asymptotic bound.

Lemma 4.18 *Let $n = |X|$. Then $|\mathsf{DTS}| = 1 + 4n^2 n!^2 \sum_{i=0}^{n} \frac{1}{i!^2} = O(n^{2n+2})$.*

Proof. By the pigeonhole principle, a dependence sequence cannot contain more than $n+1$ dependences without violating the condition of shortness.

Let $i \in \{0, \ldots, n\}$. For forming a short dependence trace $\langle d_0, \ldots, d_i \rangle$ of length $i+1$, we can choose arbitrary program variables as source variable of d_0 and as destination variable of d_i; there are n^2 ways of doing this. We can choose the remaining source variables of $d_1, \ldots, d_i$ as an arbitrary i-permutation of the variables in X. (Recall that an i-permutation of X is an ordered sequence of i elements of X, with no element appearing more than once in the sequence). The same holds for the remaining destination variables of $d_0, \ldots, d_{i-1}$. As there are $\frac{n!}{(n-i)!}$ i-permutations [6], there are thus $n^2(\frac{n!}{(n-i)!})^2$ short dependence sequences of length $i+1$. There are four possible choices for the transparency bits in a dependence trace with a given non-empty dependence sequences. In addition we have a single dependence trace with an empty dependence sequence, viz. $(0, \varepsilon, 0)$. Summing up, the number of short dependence traces is thus

$$1 + 4\sum_{i=0}^{n} \left(n^2 \left(\frac{n!}{(n-i)!} \right)^2 \right) = 1 + 4n^2 n!^2 \sum_{i=0}^{n} \frac{1}{(n-i)!^2} = 1 + 4n^2 n!^2 \sum_{i=0}^{n} \frac{1}{i!^2} .$$

Using the well-known fact that $n! \leq n^n$ and bounding the sum by

$$\sum_{i=0}^{n} \frac{1}{i!^2} \leq \sum_{i=0}^{n} \frac{1}{i!} \leq \sum_{i=0}^{\infty} \frac{1}{i!} = e$$

the asymptotic bound $O(n^{2n+2})$ follows. □

The asymptotic bound $O(n^{2n+2})$ for $|\mathsf{DTS}|$ is rather rough as it involves the rather bad estimate n^n for $n!$. Using for instance Stirling's approximation [6] for the factorial function,

$$n! = \sqrt{2\pi n} \left(\frac{n}{e} \right)^n \left(1 + \Theta\left(\frac{1}{n} \right) \right) ,$$

we could obtain tighter bounds. But the given bound suffices for our purposes.

The second reason why we are interested in short dependence traces is that they suffice to capture the potential of runs to aid in forming dependences 'up to subsumption' as the following lemma shows.

Lemma 4.19 (Short dependence traces subsume) *Suppose $r \vdash \tau$. Then there is a short dependence trace τ' with $r \vdash \tau'$ and $\tau \sqsubseteq \tau'$.*

Proof. Suppose $r \vdash \tau = (\iota, \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle, \kappa)$. We describe a shortening procedure that can be iterated until a short dependence trace is obtained.

Suppose τ is not already short. Let us assume that condition 1. is violated; if 2. is violated we can proceed analogously. Then there are indices i, j, $1 \leq i < j < k$, with $y_i = y_j$. Consider the dependence trace τ' obtained from τ by removing the middle part $\langle (x_{i+1}, y_{i+1}), \ldots, (x_j, y_j) \rangle$ of the dependence sequence:

$$\tau' := (\iota, \langle (x_1, y_1), \ldots, (x_i, y_i), (x_{j+1}, y_{j+1}), \ldots, (x_k, y_k) \rangle, \kappa) .$$

It is not hard to see that both $\tau \sqsubseteq \tau'$ and $\tau \leq \tau'$. By Proposition 4.7 the latter implies $r \vdash \tau'$. □

While this lemma shows that short dependence traces are promising, we are not yet done. We still have to see that we can obtain the short dependence traces of a composed set of runs from the short dependence traces of the argument run sets. This is particularly challenging for run sets obtained by interleaving and will be the topic of Sections 4.8–4.12.

Shortening a dependence trace w.r.t. either $\leq$ or $\sqsubseteq$ results again in a short dependence trace.

Lemma 4.20 ($\leq$ and $\sqsubseteq$ preserve shortness) *If τ is short and $\tau \leq \tau'$ or $\tau \sqsubseteq \tau'$, then τ' is short.*

Proof. All pairs of source or target variables in τ' are also pairs of target variables in τ if $\tau \leq \tau'$ or $\tau \sqsubseteq \tau'$. □

We denote the set of antichains of short dependence traces by ACS:

$$\mathsf{ACS} = \{D \in \mathsf{AC} \mid D \subseteq \mathsf{DTS}\} .$$

Lemma 4.19 implies that $\sqsubseteq$-maximal dependence traces of a run (or run set) are always short. Therefore, if we restrict attention to short dependence traces of a run or run set, we still capture all maximal dependence traces. By working with ACS instead of AC, we code this knowledge into the domain. In particular, we do not loose dependences because the dependence traces of the form $(1, \langle (a, b) \rangle, 1)$ that correspond to dependences are trivially short.

Lemma 4.21 $(\mathsf{ACS}, \sqsubseteq)$ *is a complete sub-lattice of* $(\mathsf{AC}, \sqsubseteq)$. *Its height is* $|\mathsf{DTS}| + 1 = O(n^{2n+2})$ *where* $n = |X|$.

Proof. Suppose $\mathcal{D} \subseteq \mathsf{ACS}$. In order to prove that $(\mathsf{ACS}, \sqsubseteq)$ is a complete sub-lattice of $(\mathsf{AC}, \sqsubseteq)$ we have to show that $\bigsqcup \mathcal{D} \in \mathsf{ACS}$, i.e. that $\bigsqcup \mathcal{D} \subseteq \mathsf{DTS}$:

$$\bigsqcup \mathcal{D} \underset{\substack{\uparrow \\ \text{[Lem. 4.13]}}}{=} (\bigcup \mathcal{D})^{\uparrow} \underset{\substack{\uparrow \\ \text{[Lem. 4.12]}}}{\subseteq} \bigcup \mathcal{D} \underset{\substack{\uparrow \\ [\mathcal{D} \subseteq \mathsf{ACS}]}}{\subseteq} \mathsf{DTS}\,.$$

We can restrict the downwards closure operator to short dependence traces, i.e. redefine it by $D^{\downarrow} = \{\tau \in \mathsf{DTS} \mid \exists \tau' \in D : \tau \sqsubseteq \tau'\}$ for $D \subseteq \mathsf{DTS}$. It follows as in Lemma 4.15 that $((\cdot)^{\uparrow}, (\cdot)^{\downarrow})$ is a Galois surjection from 2^{DTS} into ACS:

$$2^{\mathsf{DTS}} \underset{(\cdot)^{\downarrow}}{\overset{(\cdot)^{\uparrow}}{\rightleftarrows}} \mathsf{ACS}$$

As a consequence $(\mathsf{ACS}, \sqsubseteq)$ is isomorphic to the lattice of downwards closed subsets of DTS, ordered by set inclusion. The latter is a sub-lattice of $(2^{\mathsf{DTS}}, \subseteq)$. Hence its height (and thus the height of $(\mathsf{ACS}, \sqsubseteq)$) cannot be larger than the height of $(2^{\mathsf{DTS}}, \subseteq)$ which is $|\mathsf{DTS}| + 1$.

On the other hand, we can construct an ascending chain of size $|\mathsf{DTS}| + 1$: Let $(x_1, \ldots, x_{|\mathsf{DTS}|})$ be a topological sort of $(\mathsf{DTS}, \sqsubseteq)$, i.e., a list containing all elements of DTS such that $x_i \sqsubseteq x_j$ implies $i \le j$ for all $i, j \in \{1, \ldots, |\mathsf{DTS}|\}$. Then we can define a chain of length $\mathsf{DTS} + 1$ by choosing $A_0 = \emptyset$ and $A_i = (A_{i-1} \cup \{x_i\})^{\uparrow}$ for $i = 1, \ldots, |\mathsf{DTS}|$. $A_{i-1} \sqsubseteq A_i$ is obvious, and $A_{i-1} \neq A_i$ holds because $A_{i-1} \subseteq \{x_1, \ldots, x_{i-1}\}$, which is seen by a straightforward induction, and thus x_i is maximal in $A_{i-1} \cup \{x_i\}$ due to the topological sort property.

The asymptotic bound $|\mathsf{DTS}| + 1 = O(n^{2n+2})$ follows from Lemma 4.18. $\square$

4.7 The Abstract Domain

Let us now define the abstract domain. The values of the abstract domain are pairs (T, D) consisting of a set $T \subseteq X$ of variables and an $\sqsubseteq$-antichain D of short dependence traces. In the applications the dependence traces in D form the more interesting piece of information. T represents the variables for which a transparent run exists. This information is necessary in order to allow a proper propagation of initial and final transparency information in sequential contexts.

Thus, the abstract domain, AD, is given by

$$\mathsf{AD} = 2^X \times \mathsf{ACS}\,.$$

The order on the abstract domain, which we also denote by the symbol $\sqsubseteq$, is defined as the lift of the inclusion order on the T component and the antichain order $\sqsubseteq$ on the D component: $(T, D) \sqsubseteq (T', D')$ iff

1. $T \subseteq T'$ and

2. $D \sqsubseteq D'$.

$(\mathsf{AD}, \sqsubseteq)$ is the product lattice of the complete lattices $(2^X, \subseteq)$ and $(\mathsf{ACS}, \sqsubseteq)$ and hence also a complete lattice. Both of these lattices have $\emptyset$ as their least element. Hence, $(\emptyset, \emptyset)$ is the least element of $\sqsubseteq$.

Lemma 4.22 $(\mathsf{AD}, \sqsubseteq)$ *is a complete lattice with least element* $(\emptyset, \emptyset)$*. Its height is* $O(n^{2n+3})$ *where* $n = |X|$*.*

Proof. It only remains to prove the asymptotic bound for the height. The height of AD is the product of the height of $(2^X, \subseteq)$, which is $n+1$, and the height of $(\mathsf{ACS}, \sqsubseteq)$, which is $O(n^{2n+2})$ by Lemma 4.21. This implies the stated bound. $\square$

Let us now define an abstraction mapping $\alpha : \mathsf{NR} \to \mathsf{AD}$ that captures the intuition how non-atomic run sets are abstracted to values from AD:

$$\begin{array}{rcl} \alpha(R) & = & (T_R, D_R)\,,\ \text{where} \\ T_R & = & \{x \in X \mid \exists r \in R : r \text{ is transparent for } x\} \text{ and} \\ D_R & = & \{\tau \in \mathsf{DT} \mid \exists r \in R : r \vdash \tau\}^{\uparrow}\,. \end{array}$$

Before we proceed, let us show that this is a proper definition.

Lemma 4.23 α *is well-defined.*

Proof. We have to show two things for an arbitrary $R \in \mathsf{NR}$:

1. D_R consist of short dependence traces.

2. D_R is an $\sqsubseteq$-antichain.

To 1.: Assume there is $\tau \in D_R$ that is not short. Then there is $r \in R$ with $r \vdash \tau$. By Lemma 4.19, there is a *short* dependence trace τ' with $r \vdash \tau$ and $\tau \sqsubseteq \tau'$. In particular $\tau' \in \{\tau \in \mathsf{DT} \mid \exists r \in R : r \vdash \tau\}$ and, as τ' is short and τ is not, we even have $\tau \sqsubset \tau'$. But this shows that τ is not maximal in $\{\tau \in \mathsf{DT} \mid \exists r \in R : r \vdash \tau\}$ and hence is not a member of D_R, a contradiction.

To 2.: This is ensured by Lemma 4.12(4.). $\square$

The abstraction $\alpha(R)$ of a run set R is induced by the following abstraction $\beta(r)$ of the single runs $r \in R$:

$$\begin{array}{rcl} \beta(r) & = & (T_r, D_r)\,,\ \text{where} \\ T_r & = & \{x \in X \mid r \text{ is transparent for } x\} \text{ and} \\ D_r & = & \{\tau \in \mathsf{DT} \mid r \vdash \tau\}^{\uparrow}\,. \end{array}$$

Lemma 4.24 *Suppose* $R \in \mathsf{NR}$*. Then* $\alpha(R) = \bigsqcup\{\beta(r) \mid r \in R\}$*.*

Proof. We have $\bigsqcup\{\beta(r) \mid r \in R\} = (\bigcup_{r\in R} T_r, \bigsqcup_{r\in R} D_r)$. It is obvious that $T_R = \bigcup_{r\in R} T_r$. On the other hand, we have $\bigsqcup_{r\in R} D_r = (\bigcup_{r\in R}\{\tau \mid r \vdash \tau\}^{\uparrow})^{\uparrow}$, by Lemma 4.13. It is not hard to show that this equals D_R by considering the $\sqsubseteq$- and the $\sqsupseteq$-direction separately. □

The fact that α is induced by an abstraction on single runs has nice consequences. First of all, it immediately implies that α is monotonic.

Proposition 4.25 *α is monotonic: $R \subseteq R'$ implies $\alpha(R) \sqsubseteq \alpha(R')$.* □

Secondly, it even implies that α is universally disjunctive.

Proposition 4.26 *α is universally disjunctive.* □

This property is crucial for preciseness of the abstract interpretation of constraint systems, cf. Chapter 5, and shows us that α provides a proper abstraction of run sets by being the lower adjoint of a Galois connection. For completeness let us introduce the corresponding upper adjoint. It is $\gamma : \mathsf{AD} \to \mathsf{NR}$, defined by

$$\gamma(T, D) \;=\; \{r \mid T_r \subseteq T,\ D_r \sqsubseteq D\}\,.$$

Proposition 4.27 *(α, γ) is a Galois connection between NR and AD: $\mathsf{NR} \underset{\gamma}{\overset{\alpha}{\rightleftarrows}} \mathsf{AD}$.*
□

We leave the proof that γ is well-defined and forms a Galois connection with α to the reader.

In the sections that follow we define composition operators on AD and show that they are precise abstractions of the corresponding operators on NR. We start with the pre- and the post-operator that are rather simple. After that we discuss sequential composition. Afterwards we consider the most interesting and challenging operator: interleaving. Finally, we discuss the abstract semantics of base edges.

4.8 Pre-Operator

We define the (abstract) pre-operator, $pre^{\#} : \mathsf{AD} \to \mathsf{AD}$, as follows:

$$pre^{\#}(T, D) \;=\; \begin{cases} (\emptyset, \emptyset)\,, & \text{if } D = \emptyset \\ (X, \{(\iota, \varphi, \kappa) \in \mathsf{DT} \mid (\iota, \varphi, 0) \in D\})\,, & \text{if } D \neq \emptyset\,. \end{cases}$$

Lemma 4.28 *$pre^{\#}$ is well-defined: for any $(T, D) \in \mathsf{AD}$, $pre^{\#}(T, D) \in \mathsf{AD}$.*

Proof. The only property that is not obvious is that $A := \{(\iota, \varphi, \kappa) \in \mathsf{DT} \mid (\iota, \varphi, 0) \in D\}$ is an antichain of short dependence traces. First of all, any dependence trace $(\iota, \varphi, \kappa) \in A$ inherits being short from the dependence trace $(\iota, \varphi, 0) \in D$ that induces its inclusion in A. Secondly, assume that there are distinct dependence traces $\tau, \tau' \in A$ with $\tau \sqsubseteq \tau'$. By the definition of the subsumption order, the transparency bits in τ and τ' must coincide, i.e. we can write them in the form $\tau = (\iota, \varphi, \kappa)$ and $\tau' = (\iota, \varphi', \kappa)$. From $\tau \sqsubseteq \tau'$ it follows that also $(\iota, \varphi, 0) \sqsubseteq (\iota, \varphi', 0)$. But then $(\iota, \varphi, 0)$ and $(\iota, \varphi', 0)$ are two distinct comparable dependence traces in D, which is a contradiction to D being an antichain. Hence $pre^{\#}(T, D)$ must be an antichain of short dependence traces. □

The crucial observation for the adequacy of the definition of $pre^{\#}$ is this.

Lemma 4.29 *$r \vdash (\iota, \varphi, 0)$ if and only if there is a prefix r' of r with $r' \vdash (\iota, \varphi, \kappa)$.*

Proof. Let $\varphi = \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle$.

'$\Rightarrow$': Suppose $r \vdash (\iota, \varphi, 0)$. If $\kappa = 0$, we can choose $r' = r$. So assume $\kappa = 1$. Choose a decomposition $t_0 r_1 \cdots r_k t_k$ of r that witnesses $r \vdash (\iota, \varphi, 0)$. Let $r' = t_0 r_1 \cdots r_k$. Then, clearly, r' is a prefix of r and $t_0 r_1 \cdots r_k t'_k$ with $t'_k = \varepsilon$ is a decomposition of r' that witnesses $r' \vdash (\iota, \varphi, 1)$.

'$\Leftarrow$': Suppose r' is a prefix of r with $r' \vdash (\iota, \varphi, \kappa)$. Choose r'' with $r = r'r''$, and let $t_0 r_1 \cdots r_k t_k$ be a decomposition of r' that witnesses $r' \vdash (\iota, \varphi, \kappa)$. Then $t_0 r_1 \cdots r_k t'_k$ with $t'_k = t_k r''$ is a decomposition of r that witnesses $r \vdash (\iota, \varphi, 0)$. □

We can now show that the abstract pre-operator is a precise abstraction of the concrete pre-operator.

Theorem 4.30 (Abstract pre-operator is precise) *Suppose $R \in \mathsf{NR}$. Then $\alpha(pre(R)) = pre^{\#}(\alpha(R))$.*

Proof. If $R = \emptyset$, then $\alpha(pre(R)) = \alpha(\emptyset) = (\emptyset, \emptyset) = pre^{\#}(\emptyset, \emptyset) = pre^{\#}(\alpha(R))$. So let us assume $R \neq \emptyset$.

By unfolding the definitions, we see that $\alpha(pre(R)) = (T_{pre(R)}, D_{pre(R)})$ with

$$\begin{aligned} T_{pre(R)} &= \{x \mid \exists r, r' \in \mathsf{Runs} : r \cdot r' \in R \wedge r \text{ is transparent for } x\} \\ D_{pre(R)} &= \{\tau \mid \exists r, r' \in \mathsf{Runs} : r \cdot r' \in R \wedge r \vdash \tau\}^{\uparrow}. \end{aligned}$$

In order to evaluate the right hand side, note first that D_R is non-empty: there is a run $r \in R$ and any such run satisfies $r \vdash (0, \varepsilon, 0)$; moreover, $(0, \varepsilon, 0)$ is $\sqsubseteq$-maximal and hence contained in D_R. Consequently, the second case applies in the definition of $pre^{\#}$ and we have $pre^{\#}(\alpha(R)) = pre^{\#}(T_R, D_R) = (X, D)$ with

$$D = \{(\iota, \varphi, \kappa) \in \mathsf{DT} \mid (\iota, \varphi, 0) \in D_R\}^{\uparrow}$$

Thus, we have to show $T_{pre(R)} = X$ and $D_{pre(R)} = D$.

$T_{pre(R)} \subseteq X$ is trivial. In order to see the reverse inclusion, i.e. that $T_{pre(R)}$ contains any $x \in X$, choose an arbitrary $r \in R$ and observe that the empty run ε is a prefix of r that is transparent for any variable x.

The following chain of implications shows $D_{pre(R)} \sqsubseteq D$:

$$
\begin{array}{ll}
& (\iota, \varphi, \kappa) \in D_{pre(R)} \\
\Rightarrow & [\text{Equation above, Lemma 4.12(1.)}] \\
& \exists r, r' \in \mathsf{Runs} : r \cdot r' \in R \wedge r \vdash (\iota, \varphi, \kappa) \\
\text{iff} & [\text{Lemma 4.29}] \\
& \exists r \in R : r \vdash (\iota, \varphi, 0) \\
\text{iff} & [\text{Set comprehension}] \\
& (\iota, \varphi, 0) \in \{\tau \in \mathsf{DT} \mid \exists r \in R : r \vdash \tau\} \\
\Rightarrow & [\text{Lemma 4.11, definition } D_R] \\
& \exists \tau' \in D_R : (\iota, \varphi, 0) \sqsubseteq \tau' \\
\Rightarrow & [\text{See below}] \\
& \exists \tau \in D : (\iota, \varphi, \kappa) \sqsubseteq \tau \, .
\end{array}
$$

The reasoning for the last step is as follows. The fewer gaps ordering $\sqsubseteq$ is concerned only with removing gaps from the dependence sequence φ in a dependence trace but leaves the initial and final transparency information untouched. Hence, the dependence trace $\tau' \in D_R$ with $(\iota, \varphi, 0) \sqsubseteq \tau'$ must have the form $\tau' = (\iota, \psi, 0)$. But then $\tau := (\iota, \psi, \kappa) \in D$ and $(\iota, \varphi, \kappa) \sqsubseteq (\iota, \psi, \kappa)$.

Finally, we show $D \sqsubseteq D_{pre(R)}$:

$$
\begin{array}{ll}
& (\iota, \varphi, \kappa) \in D \\
\Rightarrow & [\text{Above equation for } D, \text{Lemma 4.12(1.)}] \\
& (\iota, \varphi, 0) \in D_R \\
\Rightarrow & [\text{Definition of } D_R, \text{Lemma 4.12(1.)}] \\
& \exists r \in R : r \vdash (\iota, \varphi, 0) \\
\text{iff} & [\text{Lemma 4.29}] \\
& \exists r, r' : r \cdot r' \in R \, \wedge \, r \vdash (\iota, \varphi, \kappa) \\
\text{iff} & [\text{Set comprehension}] \\
& (\iota, \varphi, \kappa) \in \{\tau \in \mathsf{DT} \mid \exists r, r' : r \cdot r' \in R \, \wedge \, r \vdash \tau\} \\
\Rightarrow & [\text{Lemma 4.11}] \\
& \exists \tau \in \{\tau \in \mathsf{DT} \mid \exists r, r' : r \cdot r' \in R \, \wedge \, r \vdash \tau\}^{\uparrow} : (\iota, \varphi, \kappa) \sqsubseteq \tau \\
\text{iff} & [\text{Above equation for } D_{pre(R)}] \\
& \exists \tau \in D_{pre(R)} : (\iota, \varphi, \kappa) \sqsubseteq \tau \, .
\end{array}
$$

This completes the proof. $\square$

4.9 Post-Operator

We define the (abstract) post-operator, $post^{\#} : \mathsf{AD} \to \mathsf{AD}$, in complete analogy to the pre-operator as follows:

$$post^{\#}(T, D) \;=\; \begin{cases} (\emptyset, \emptyset)\,, & \text{if } D = \emptyset \\ (X, \{(\iota, \varphi, \kappa) \in \mathsf{DT} \mid (0, \varphi, \kappa) \in D\})\,, & \text{if } D \neq \emptyset\,. \end{cases}$$

By symmetry to the pre-operator we obtain that the post operator is well-defined and a precise abstraction of the post-operator on non-atomic run sets.

Theorem 4.31 (Abstract post-operator is precise) *Suppose $R \in \mathsf{NR}$. Then $\alpha(post(R)) = post^{\#}(\alpha(R))$.* $\square$

4.10 Sequential Composition

The *(abstract) sequential composition operator*, $;^{\#} : \mathsf{AD} \times \mathsf{AD} \to \mathsf{AD}$, which we write as an infix operator, is defined by

$$(T, D);^{\#}(T', D') \;=\; (T \cap T', (D \cdot D')^{\uparrow})\,,$$

where

$$\begin{aligned} D \cdot D' \;=\; & \{(\iota, \varphi, \kappa) \in D \mid \kappa = 1 \Rightarrow \overrightarrow{\varphi} \in T'\} && (4.1)\\ & \cup \{(\iota, \varphi, \kappa) \in D' \mid \iota = 1 \Rightarrow \overleftarrow{\varphi} \in T\} && (4.2)\\ & \cup \{(\iota, \varphi \cdot \psi, \kappa) \in \mathsf{DTS} \mid (\iota, \varphi, 0) \in D, (0, \psi, \kappa) \in D'\} && (4.3)\\ & \cup \{(\iota, \varphi \cdot (x, z) \cdot \psi, \kappa) \in \mathsf{DTS} \mid && (4.4)\\ & \qquad \exists y : (\iota, \varphi \cdot (x, y), 1) \in D, (1, (y, z) \cdot \psi, \kappa) \in D'\}\,. \end{aligned}$$

Before we explain the intuition underlying this definition we show well-definedness.

Lemma 4.32 *The abstract sequential composition operator $;^{\#}$ is well-defined.*

Proof. We have to show that $(D \cdot D')^{\uparrow} \in \mathsf{ACS}$ for all $D, D' \in \mathsf{ACS}$, i.e. that $(D \cdot D')^{\uparrow}$ is an $\sqsubseteq$-antichain of short dependence traces.

It is easy to see that $D \cdot D'$ (and hence its subset $D \cdot D')^{\uparrow}$) contains only short dependence traces: the first two sets contain only dependence traces from D or D', which consequently are short, and the constructions in the third and forth set are explicitly restricted to contain short dependence traces. The application of the $^{\uparrow}$-operator ensures that $(D \cdot D')^{\uparrow} \in \mathsf{ACS}$ is an $\sqsubseteq$-antichain. $\square$

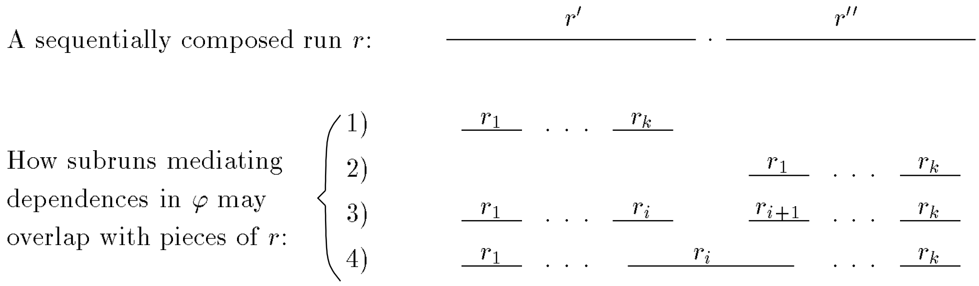


Figure 4.2: Intuition of sequential composition.

Obviously, a run $r = r' \cdot r''$ composed of two runs r' and r'' is transparent for a variable x if and only if both r' and r'' are. Therefore, transparency information must be intersected in a sequential composition.

Let us explain the intuition underlying the definition of $D \cdot D'$. Suppose given a run $r = r' \cdot r''$ which is composed of two runs $r' \in D$ and $r'' \in D'$ that use distinct virtual variables ($\mathsf{virtual}(r') \cap \mathsf{virtual}(r'') = \emptyset$). Assume that $\tau = (\iota, \varphi, \kappa)$ with $\varphi = \langle d_1, \dots, d_k \rangle$ is a dependence trace compatible with r. Each dependence d_i in φ is mediated by a sub-piece r_i of r; we can choose the r_i as short as possible (i.e., such that it starts with an assignment that reads the source variable of d_i and ends with an assignment to the destination variable of d_i). There are four possibilities, how these sub-pieces may lie in r as illustrated in Fig. 4.2:

1) all of them lie in r';

2) all of them lie in r'';

3) there is an i, $1 \leq i < k$, such that $r_1, \dots, r_i$ lie in r' and $r_{i+1}, \dots, r_k$ lie in r'';

4) there is an i such that r_i overlaps with the 'join point' of r' and r''.

These four cases are handled by the four sets appearing in the definition of $D \cdot D'$:

1) in this case, τ is also a dependence trace of r'. Vice versa, dependence traces $\tau' = (\iota', \phi', \kappa')$ of r' give rise to dependence traces of r. However, if $\kappa' = 1$, no statement that kills $\overrightarrow{\varphi'}$, the destination variable of the last dependence in φ', is allowed after r_k. Therefore, r' must be transparent for $\overrightarrow{\varphi'}$; hence the side condition in set (4.1).

2) this case is symmetric to case 1).

3) in this case, r' has the dependence trace $(\iota, \langle d_1, \dots, d_i \rangle, 0)$ and r'' the dependence trace $(0, \langle d_{i+1}, \dots, d_k \rangle, \kappa)$. Vice versa, dependence traces of r' and r'' of this form give rise to a dependence trace of r.

4) choose variables $x, z \in X$ such that $d_i = (x, z)$. Sub-run r_i accomplishes the transfer from x to z via certain intermediate variables. One of these intermediate variables, say y, must bridge the joint point between r' and r'' (i.e., it is assigned to in r', read from in r'' and not killed in between). As r and r' use distinct virtual variables, y must be a program variable: $y \in X$. Then $\langle s, \langle d_1, \ldots, d_{i-1}, (x, y)\rangle, 1\rangle$ is a dependence of r' and $\langle 1, \langle d_1, \ldots, d_{i-1}, (x, y)\rangle, 1\rangle$ is a dependence of r''. 1 as the final component of τ' and as the first component of τ'' is justified, as y is not killed from the place where it is assigned to in r' and read in r''. Similarly, dependences of r' and r'' of the above form give rise to a dependence trace of r.

It is not hard to see that in all four cases the dependence traces of r' and/or r'' in question are short and $\sqsubseteq$-maximal if τ is and, vice versa, that each short and $\sqsubseteq$-maximal dependence trace of r can be composed of short and $\sqsubseteq$-maximal dependence traces compatible with r' and r'' in the described way.

Lemma 4.33 (Abstract sequential composition operator is precise) *Suppose* $R, S \in \mathsf{NR}$. *Then* $\alpha(R\,;S) = \alpha(R);^{\#}\alpha(S)$.

Proof. By formalizing the intuition described above. □

4.11 Interleaving

Transparency information for the interleaving $R \otimes S$ of two run sets R and S is easy to obtain from transparency information of the components: a transparent run for a variable x exists in $R \otimes S$ if and only if each component set contains a transparent run. Therefore, the transparency information in T_R and T_S must simply be intersected.

By far more interesting is to consider the dependence traces in $D_{R\otimes S}$ as the two threads modeled by R and S can cooperate in order to mediate dependences. More specifically, a dependence (u, v) can be composed of complementary dependence sequences of two runs $r \in R$ and $s \in S$, e.g., as illustrated here:

Transfers of r:	$u = x_1 \to y_1$	$x_2 \to y_2$	$x_3 \to y_4$	$\cdots$	$x_{k-1} \to y_{k-1}$	$x_k \to y_k = v$
Transfers of s:		$y_1 \to x_2$	$y_2 \to x_3$	$\cdots$	$y_{k-1} \to x_k$	

Of course such a combination of complementary dependence sequences can also start and/or end with a dependence of s. And, as a border case, one of the dependence sequences can be empty; the other then just consists of a single dependence. Before we define the abstract interleaving operator, we present in the next section the general definition of when two dependence traces complement each other to a single dependence and also introduce a relation C that extends this definitions to dependence traces.

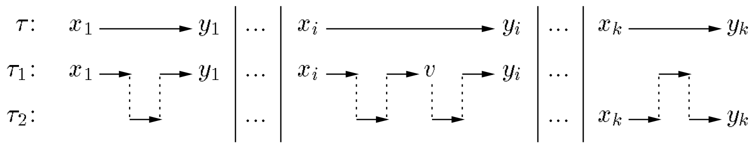


Figure 4.3: Complementary dependence traces.

4.11.1 Complementary Dependence Traces

Let $\varphi, \psi \in \mathsf{DS}$ be two dependence sequences (one of them can be empty) and $u, v \in X$. Choose variables such that $\varphi = \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle$, $k \geq 0$. We say that ψ *complements* φ *to* (u, v) if one of the following cases applies:

1. $\varphi \neq \varepsilon$, $u = \overleftarrow{\varphi}$, $v = \overrightarrow{\varphi}$, and $\psi = \langle (y_1, x_2), \ldots, (y_{k-1}, x_k) \rangle$;

2. $\varphi \neq \varepsilon$, $\psi \neq \varepsilon$, $u = \overleftarrow{\varphi}$, $v = \overrightarrow{\psi}$, and $\psi = \langle (y_1, x_2), \ldots, (y_{k-1}, x_k), (y_k, v) \rangle$;

3. $\varphi \neq \varepsilon$, $\psi \neq \varepsilon$, $u = \overleftarrow{\psi}$, $v = \overrightarrow{\varphi}$, and $\psi = \langle (u, x_1), (y_1, x_2), \ldots, (y_{k-1}, x_k) \rangle$; or

4. $\psi \neq \varepsilon$, $u = \overleftarrow{\psi}$, $v = \overrightarrow{\psi}$, and $\psi = \langle (u, x_1), (y_1, x_2), \ldots, (y_{k-1}, x_k), (y_k, v) \rangle$.

Intuitively, ψ complements φ to (u, v) if the two of them can alternately be combined to a gap-free transfer from u to v. The different cases are distinguished by whether the first read in this gap-free transfer comes from φ (cases 1/2) or ψ (cases 3/4) and whether the last write is in φ (cases 1/3) or ψ (cases 2/4).

Now, consider a dependence trace τ compatible with a run $t \in R \otimes S$ which is an interleaving of the runs $r \in R$, $s \in S$. Then every single dependence in τ must be obtained in the above described fashion from pieces of dependence traces compatible with r and s. We, therefore, generalize this notion of completion to dependence traces as follows: Suppose given dependence traces τ, τ_0, τ_1, where $\tau = (\iota, \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle, \kappa)$, $\tau_0 = (\iota_0, \varphi, \kappa_0)$, $\tau_1 = (\iota_1, \psi, \kappa_1)$. Then we say that τ_1 complements τ_0 to τ, $C(\tau_0, \tau_1, \tau)$ for short, if there are dependence sequences $\varphi_1, \ldots, \varphi_k, \psi_1, \ldots, \psi_k$ such that

1. $\varphi = \varphi_1 \cdot \ldots \cdot \varphi_k$ and $\psi = \psi_1 \cdot \ldots \cdot \psi_k$;

2. ψ_i complements ϕ_i to (x_i, y_i) for $i = 1, \ldots, k$.

3. $\iota = 1$ implies $\iota_0 = 1$ and ψ_1 complements φ_1 to (x_1, y_1) according to cases 1 and 2, or $\iota_1 = 1$ and ψ_1 complements φ_1 according to cases 3 and 4; and

4. $\kappa = 1$ implies $\kappa_0 = 1$ and ψ_k complements φ_k to (x_k, y_k) according to cases 1 and 3, or $\kappa_1 = 1$ and ψ_k complements φ_k according to cases 2 and 4.

The typical situation of two dependence traces τ_0 and τ_1 that complement each other to a dependence trace τ is illustrated in Fig. 4.3. For clarity we omit the transparency bits. The dashed vertical lines indicate equality of variables.

A number of elementary properties of the relation C is collected in the following lemma.

Lemma 4.34 (Basic properties of C) *Suppose $\tau, \tau_0, \tau_1 \in \mathsf{DT}$. Then*

1. *C is symmetric in the first two parameters: $C(\tau_0, \tau_1, \tau)$ if and only if $C(\tau_1, \tau_0, \tau)$.*
2. *$(0, \varepsilon, 0)$ is a 'neutral element': $C((0, \varepsilon, 0), \tau, \tau)$.*
3. *In particular, $C((0, \varepsilon, 0), (0, \varepsilon, 0), (0, \varepsilon, 0))$.*

Proof. Left to the reader. □

4.11.2 Interleaving Operator

We are now in the position to define the *(abstract) interleaving operator*, $\otimes^{\#}$: $\mathsf{AD} \times \mathsf{AD} \to \mathsf{AD}$, which we write again as an infix operator:

$$(T, D) \otimes^{\#} (T', D') \;=\; (T \cap T', \{\tau'' \in \mathsf{DTS} \mid \exists \tau \in D, \tau' \in D' : C(\tau, \tau', \tau'')\}^{\uparrow}) .$$

By restricting the set construction to short dependence traces and applying the $(\cdot)^{\uparrow}$ operator, the interleaving operator is trivially well-defined. The goal of the remainder of this section is to show that it is a precise abstraction of the interleaving operator on sets of non-atomic runs.

Theorem 4.35 (Abstract interleaving operator is precise)
Suppose $R, S \in \mathsf{NR}$. Then $\alpha(R \otimes S) = \alpha(R) \otimes^{\#} \alpha(S)$.

The proof is deferred to Section 4.11.5. Before that, we establish a number of lemmas that capture the main insights underlying the proof.

4.11.3 Soundness Lemmas

The lemmas in this section are concerned with the soundness of the abstract interleaving composition operator, i.e. they are crucial for the proof that $\alpha(R \otimes S) \sqsubseteq \alpha(R) \otimes^{\#} \alpha(S)$ for any two run sets R, S. The critical point here is to guarantee that our definition of the abstract interleaving operator includes enough dependence traces.

As a first step, we show that each dependence trace that is compatible with some interleaving of two runs r, s can also be obtained by combining two dependence traces of the component runs r and s via the relation C.

Let $r, s, t \in \mathsf{Runs}$ with $\mathsf{virtual}(r) \cap \mathsf{virtual}(s) = \emptyset$ and $\tau \in \mathsf{DT}$.

Lemma 4.36 *Suppose $t \in r \otimes s$ and $t \vdash \tau$. Then there are $\tau_r, \tau_s \in \mathsf{DT}$ with $r \vdash \tau_r$, $s \vdash \tau_s$, and $C(\tau_r, \tau_s, \tau)$.*

Proof. Assume that t is an interleaving of r and s and $\tau = (\iota, \langle d_1, \ldots, d_k\rangle, \kappa)$ is a dependence trace of t. Each dependence d_i is mediated by a certain sub-run t_i of t and each t_i is an interleaving of certain sub-runs of r and s.

From t_i we can construct dependence traces φ_i and ψ_i of these sub-pieces of r and s such that φ_i complements ψ_i to dependence d_i. This is described below. Then $\varphi_1 \cdot \ldots \cdot \varphi_k$ and $\psi_1 \cdot \ldots \cdot \psi_k$ are dependence sequences of r and s, resp., and we can choose transparency bits $\iota_r, \kappa_r, \iota_s, \kappa_s \in \mathbb{B}$ such that $\tau_r = (\iota_r, \varphi_1 \cdot \ldots \cdot \varphi_k, \kappa_r)$ and $\tau_s = (\iota_s, \psi_1 \cdot \ldots \cdot \psi_k, \kappa_s)$ are dependence traces of r and s, resp., such that $C(\tau_r, \tau_s, \tau)$ holds. Specifically, we choose $\iota_r = \iota$ if the first assignment instance involved in the mediation of d_1 belongs to r and $\iota_s = \iota$ if it belongs to s, and similarly for the final transparency bits and the last assignment instance involved in the mediation of d_k. All other transparency bits are chosen 0.

Let us now explain how to construct the dependence sequences φ_i and ψ_i mentioned above. Choose program variables x, y such that $d_i = (x, y)$. Sub-run t_i of t mediates d_i via certain assignment instances $a_j := e_j$, $j = 1, \ldots, l$, as specified in the definition of mediation. In particular, $a_l = y$. Each of these assignment instances lies either in a sub-piece of r or a sub-piece of s. Let us consider the case that the first assignment instance $a_1 := e_1$ lies in a sub-piece of r; the case that it lies in a sub-piece of r is analogous. We can then find indices $0 < j_0 < j_1 < \ldots < j_n$ such that $a_j := e_j$ lies in a sub-piece of r if $j_m < j \leq j_{m+1}$ for an *even* $m \in \{0, \ldots, n-1\}$ and in a sub-piece of s otherwise. In particular, for any $j \in \{j_1, \ldots, j_{n-1}\}$ one of the assignments instances $a_j := e_j$ and $a_{j+1} := e_{j+1}$ lies in a sub-piece of r and the other one in a sub-piece of s. This implies that a_j must be a program variable, because it appears in e_{j+1} according to the definition of mediation and $\mathsf{virtual}(r) \cap \mathsf{virtual}(s) = \emptyset$. Choose now

$$\begin{aligned}\varphi_i &= \langle (x, a_{j_1}), (a_{j_2}, a_{j_3}), \ldots, (a_{j_{n-2}}, a_{j_{n-1}})\rangle\,,\\ \psi_i &= \langle (a_{j_1}, a_{j_2}), (a_{j_3}, a_{j_4}), \ldots, (a_{j_{n-1}}, y)\rangle\end{aligned}$$

if n is even and

$$\begin{aligned}\varphi_i &= \langle (x, a_{j_1}), (a_{j_2}, a_{j_3}), \ldots, (a_{j_{n-1}}, y)\rangle\,,\\ \psi_i &= \langle (a_{j_1}, a_{j_2}), (a_{j_3}, a_{j_4}), \ldots, (a_{j_{n-2}}, a_{j_{n-1}})\rangle\end{aligned}$$

if n is odd. Then φ_i and ψ_i are dependence sequences of the sub-runs of r and s that comprise t_i and, obviously, φ_i complements ψ_i to d_i. $\square$

Example 4.37 *Fig. 4.4 illustrates the construction in the proof of Lemma 4.36. The run t is an interleaving of the runs r and s. We can thus decompose r*

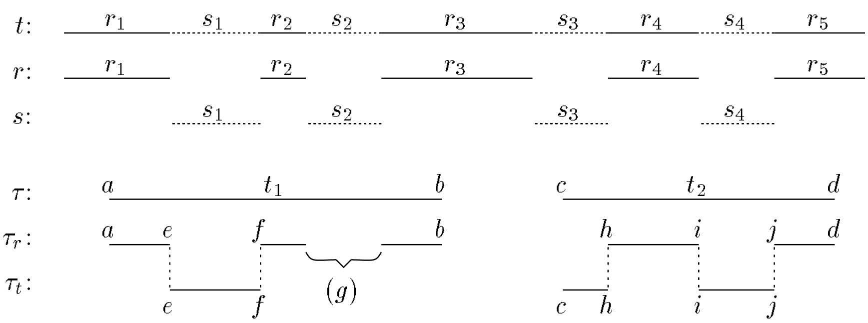


Figure 4.4: Dependence traces of interleavings are induced by complementary dependence traces of the components.

and s into sub-runs such that t is obtained by alternately shuffling these sub-runs together; in the example $r = r_1r_2r_3r_4r_5$, $s = s_1s_2s_3s_4$, and $t = r_1s_1r_2s_2r_3s_3r_4s_4r_5$.

Let us assume that $\tau = (\iota, \langle(a,b),(c,d)\rangle, \kappa)$ is a dependence trace of t. Then there are sub-runs t_1 and t_2 of t that mediate the two dependences (a,b) and (c,d), e.g., as shown in the figure. These sub-runs overlap in a certain way with the decompositions of r and s; in the example in the figure, for instance, t_1 overlaps with a postfix of r_1, all of s_1, r_2, s_2, and a prefix of r_3. The dependence (a,b) is mediated via certain intermediate assignments $a_i := e_i$ (not shown in the figure); we call these assignments crucial *in the following.*

There may be sub-runs of r and/or s that overlap with t_i but do not contain a crucial assignment. Such sub-runs must be transparent for the variable that transfers the dependence at this moment and can be ignored. In our example, r_2 is such a sub-run and g is the variable that transfers the dependence while r_2 is executed.

Whenever two successive crucial assignments lie in sub-pieces of different runs, the dependence must be transfered in a program variable between these assignments because r and s do not share virtual variables. In the figure, e.g., e is the variable that transfers the dependence from the last crucial assignment in r_1 to the first crucial assignment in s_1 and f transfers it from the last crucial assignment in s_1 to the first crucial assignment in r_2. From these variables we can construct dependence traces τ_r and τ_s of r and s such that $C(\tau_r, \tau_s, \tau)$ holds. In Fig. 4.4, for instance, we have $\tau_r = (\iota, \langle(a,e),(f,b),(h,i),(j,d)\rangle, \kappa)$ and $\tau_s = (0, \langle(e,\kappa),(c,h),(i,j)\rangle, 0)$. □

Lemma 4.36 ensures that combining dependence traces of component runs via C is fundamentally rich enough to give us all dependence traces of potential interleavings. However, in our abstract domain, we do not collect *all* dependence traces but only the *maximal* ones. Therefore, we only combine the maximal dependence traces of component runs in the definition of interleaving, which is the best we can do with the available information. A legitimate question to ask

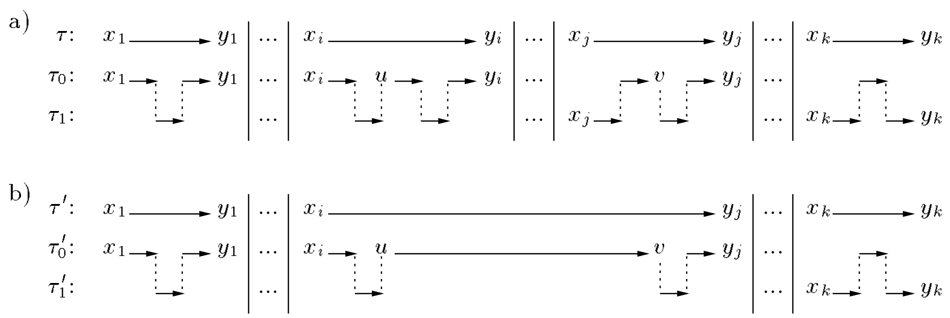


Figure 4.5: Removing gaps in a component dependence trace

is, whether this is sufficient. Can we really obtain all the *maximal* dependence traces just from the *maximal* dependence traces of the components? The answer to this question must be yes; otherwise Theorem 4.35 would be wrong. But Lemma 4.36 alone does not suffice to prove this.

The next lemma provides a kind of shortening rule that is crucial for the proof that maximal dependence traces of component run sets suffice to infer the maximal dependence traces of their interleaving.

Suppose $\tau_0, \tau_0', \tau_1, \tau \in \mathsf{DT}$.

Lemma 4.38 *Suppose $C(\tau_0, \tau_1, \tau)$ and $\tau_0 \sqsubseteq \tau_0'$. Then there are dependence traces $\tau_1', \tau' \in \mathsf{DT}$ such that $\tau_1 \leq \tau_1'$, $\tau \sqsubseteq \tau'$, and $C(\tau_0', \tau_1', \tau')$. By symmetry of C (Lemma 4.34(1)), an analogous property holds with the roles of τ_0 and τ_1 exchanged.*

Proof. The proof is illustrated in Fig. 4.5. In a) the typical situation of dependence traces τ_0, τ_1 and τ with $C(\tau_0, \tau_1, \tau)$ is shown. For clarity the transparency bits are omitted. In b) a typical dependence trace τ_0' with $\tau_0 \sqsubseteq \tau_0'$ is shown. It is obtained from τ_0 by removing all gaps between the target variable u of a certain dependence d in τ_0 and the destination variable v of a later dependence e. We can remove all the dependences from τ_1 that are used to fill some or all of these gaps in $C(\tau_0, \tau_1, \tau)$. This results in a dependence trace τ_1' with $\tau_1 \leq \tau_1'$ as shown in b). Then the dependence traces τ_0' and τ_1' complement each other to a dependence trace τ' with $\tau \leq \tau'$ as shown. As border cases, we may have $\tau_0' = \tau_0$, if none of the gaps between d and e is filled in $C(\tau_0, \tau_1, \tau)$, or $\tau' = \tau$ if d and e are used in $C(\tau_0, \tau_1, \tau)$ in the same dependence of τ. But this does not invalidate our reasoning as $\sqsubseteq$ and $\leq$ are reflexive. $\square$

By iteratively applying this shortening rule, we obtain the following lemma that is of direct use in the proof of Theorem 4.35.

Lemma 4.39 *Suppose $\tau_r \in \{\tau \mid \exists r \in R : r \vdash \tau\}$, $\tau_s \in \{\tau \mid \exists s \in S : s \vdash \tau\}$, and $C(\tau_r, \tau_s, \tau)$. Then there are $\tau'_r \in D_R$, $\tau'_s \in D_S$, and $\tau' \in \mathsf{DT}$ with $C(\tau'_r, \tau'_s, \tau')$ and $\tau \sqsubseteq \tau'$.*

Proof. The problem is that τ_r and τ_s need not be $\sqsubseteq$-maximal in their respective set. Hence they may not belong to D_R and D_S, respectively. By iteratively applying Lemma 4.38, however, we can determine dependence traces $\tau_r^\uparrow$ and $\tau_s^\uparrow$ that are $\sqsubseteq$-maximal in these sets (and hence belong to D_R and D_S, respectively) as well as a dependence trace $\tau^\uparrow$ with $C(\tau_r^\uparrow, \tau_s^\uparrow, \tau^\uparrow)$ and $\tau \sqsubseteq \tau^\uparrow$:

We start with $(\tau_r^\uparrow, \tau_s^\uparrow, \tau^\uparrow) := (\tau_r, \tau_s, \tau)$. This initialization trivially ensures $\tau_r^\uparrow \in \{\tau \mid \exists r \in R : r \vdash \tau\}$, $\tau_s^\uparrow \in \{\tau \mid \exists s \in S : s \vdash \tau\}$, $C(\tau_r^\uparrow, \tau_s^\uparrow, \tau^\uparrow)$, and $\tau \sqsubseteq \tau^\uparrow$, which is an invariant of the loop we describe in the following.

If $\tau_r^\uparrow$ is not $\sqsubseteq$-maximal in $\{\tau \mid \exists r \in R : r \vdash \tau\}$, we can choose a dependence trace $\tau'_r \in \{\tau \mid \exists r \in R : r \vdash \tau\}$ which is strictly larger: $\tau_r^\uparrow \sqsubset \tau'_r$. Then, by Lemma 4.38, there are τ'_s and τ' with $\tau_s^\uparrow \leq \tau'_s$, $\tau \sqsubseteq \tau^\uparrow \sqsubseteq \tau'$, and $C(\tau'_r, \tau'_s, \tau')$. By Proposition 4.7, $\tau'_s \in \{\tau \mid \exists r \in R : r \vdash \tau\}$, hence the invariant remains valid. We then set $(\tau_r^\uparrow, \tau_s^\uparrow, \tau^\uparrow) := (\tau'_r, \tau'_s, \tau')$. We can proceed analogously, if $\tau_s^\uparrow$ is not maximal in $\{\tau \mid \exists s \in S : s \vdash \tau\}$.

This shortening procedure is applied iteratively until both $\tau_r^\uparrow$ and $\tau_s^\uparrow$ are $\sqsubseteq$-maximal in their respective sets. Termination is guaranteed, because in each step either the dependence sequence in $\tau_r^\uparrow$ or in $\tau_s^\uparrow$ becomes shorter and the dependence sequence in the other dependence trace does not become longer. □

4.11.4 Completeness Lemmas

The lemmas in this section are concerned with completeness of the interleaving operator, i.e. they are important for the proof that $\alpha(R \otimes S) \sqsupseteq \alpha(R) \otimes^\# \alpha(S)$ for any two non-atomic run sets R, S. They crucially depend on runs being non-atomic.

A dependence mediated by a non-atomic run r must involve a virtual variable at a certain stage as assignments that have program variables on both the left- and the right-hand-side do not occur in non-atomic runs. But when the execution of r is in such a stage, no parallel thread can disturb propagation of the dependence because parallel threads do not interfere on virtual variables. This is the idea underlying the proof of the following lemma.

Lemma 4.40 *Suppose r, s are runs with $\mathsf{virtual}(r) \cap \mathsf{virtual}(s) = \emptyset$, and $x, y \in X$. If r mediates (x, y) then there is a run $t \in r \otimes s$ that mediates (x, y).*

Proof. Suppose r mediates (x, y). This means that r can be written in the form $r = r_0 \cdot \langle a_1 := e_1 \rangle \cdot r_1 \cdot \langle a_2 := e_2 \rangle \cdot r_2 \cdot \ldots \cdot r_{l-1} \cdot \langle a_l := e_l \rangle \cdot r_l$ as in the definition of mediation. Then in particular e_1 contains the variable x. As x is a program variable, this implies by the form of assignments appearing in runs that a_1 must be a virtual variable (cf. the definition of Asg). As $\mathsf{virtual}(r) \cap \mathsf{virtual}(s) = \emptyset$, s therefore cannot contain an assignment to a_1. Consequently, s is transparent for a_1. Hence the run $t \in r \otimes s$ defined by

$$t := r_0 \cdot \langle a_1 := e_1 \rangle \cdot s \cdot r_1 \cdot \langle a_2 := e_2 \rangle \cdot r_2 \cdot \ldots \cdot r_{l-1} \cdot \langle a_l := e_l \rangle \cdot r_l$$

still mediates the dependence (x, y). $\square$

Note that this argument crucially depends on the assumption about the form of assignments in runs that derives from the assumption that assignments execute non-atomically. If assignments execute atomically, the above lemma is no longer valid.

Example 4.41 *Consider the parallel execution of the two straight-line programs* $\pi_1 = (y := x)$ *and* $\pi_2 = (x := 0\,; y := 0)$.

If assignment statements execute atomically, there are just three possible runs,

1) $\langle x := 0, y := 0, y := x \rangle$,

2) $\langle x := 0, y := x, y := 0 \rangle$, *and*

3) $\langle y := x, x := 0, y := 0 \rangle$.

None of these runs mediates the dependence (x, y) *because either* x *is killed before* $y := x$ *is executed as in 1) and 2), or* y *is killed after* $y := x$ *is executed as in 2) and 3).*

If, on the other hand, assignment statements may execute non-atomically, then the two initialization statements in π_2 *could well be executed after* x *is read but before* y *is written. This is witnessed by the run*

4 $\langle v := x, x := 0, y := 0, y := v \rangle$,

where v *is a virtual variable, in our model of non-atomic execution. In contrast to the runs 1)-3), run 4) mediates the dependence* (x, y). $\square$

Lemma 4.40 provides an intuitive explanation why precise analysis of parallel programs is simpler if we assume non-atomic execution of assignments. With this assumption dependences once generated by a thread cannot be definitely destroyed by its environment. Thus, an analysis that collects positive information about potential dependences is precise. (In order to do so in a compositional fashion though it must collect more information, namely (maximal, short) dependence traces.)

This is different if we analyze with respect to the assumption that assignments execute atomically. Then there is a complex interplay between the way dependences are generated by a thread and the order of re-initializations performed by its environment as illustrated by the above example. Therefore, an analysis that just collects positive information is doomed to be imprecise.

Lemma 4.42 *Suppose r_0, r_1 are runs with* $\mathsf{virtual}(r_0) \cap \mathsf{virtual}(r_1) = \emptyset$ *and τ_0, τ_1, τ are dependence traces with $r_0 \vdash \tau_0$, $r_1 \vdash \tau_1$, and $C(\tau_0, \tau_1, \tau)$. Then there is a run $r \in r_0 \otimes r_1$ such that $r \vdash \tau$.*

Proof. For notational convenience, we discuss the case that the dependence sequence in τ consists of just a single transfer; the generalization to arbitrary transfer sequences is left to the reader. Let $\tau = (\iota, \langle (u, v) \rangle, \kappa)$. Furthermore, let $\tau_0 = (\iota_0, \phi, \kappa_0)$ and $\tau_1 = (\iota_1, \psi, \kappa_1)$.

Let us assume that case 2 in the definition of $C(\tau_0, \tau_1, \tau)$ applies; the other cases are similar. Then we can choose variables $u = x_1, \ldots, x_{k+1} = v$ such that

$$\begin{aligned} \varphi &= \langle (x_1, y_1), \ldots, (x_k, y_k) \rangle, \\ \psi &= \langle (y_1, x_2), \ldots, (y_k, x_{k+1}) \rangle, \end{aligned}$$

and it is $\iota_0 = 1$ if $\iota = 1$ and $\kappa_1 = 1$ if $\kappa = 1$. As $r_0 \vdash \tau_0$ and $r_1 \vdash \tau_1$ we can write r_0 and r_1 in the form

$$\begin{aligned} r_0 &= t_0^0 r_1^0 t_1^0 r_2^0 \cdots t_{k-1}^0 r_k^0 t_k^0 \\ r_1 &= t_0^1 r_1^1 t_1^1 r_2^1 \cdots t_{k-1}^1 r_k^1 t_k^1 \end{aligned}$$

such that

a) r_i^0 mediates (x_i, y_i) and r_i^1 mediates (y_i, x_{i+1}) for $i = 1, \ldots, k$;

b) t_0^0 is transparent for u if $\iota = 1$ (and hence $\iota_0 = 1$); and

c) t_k^1 is transparent for v if $\kappa = 1$ (and hence $\kappa_1 = 1$).

The run $r_1^0 r_1^1 r_2^0 r_2^1 \cdots r_k^0 r_k^1$ clearly mediates the dependence (u, v), but in order to construct an interleaving of r_0 and r_1, we must also execute the intermediate code pieces t_i^j. Fortunately, each of the dependences realized by some r_i^j must involve a virtual variable; and, while the transfer is in such a stage, code pieces of the other run, r_{1-j}, can safely be executed without destroying the dependence, due to the disjointness of the virtual variables used in r_0 and r_1. Thus, we can execute each code piece t_i^1 at such a stage of execution of r_{i+1}^0 and, similarly, t_i^0 during such a stage of r_i^1. The rest of the proof pursues this argument more formally.

By Lemma 4.40, there are interleavings $s_i^0 \in r_i^0 \otimes t_{i-1}^1$ and $s_i^1 \in r_i^1 \otimes t_i^0$ such that, for $i = 1, \ldots, k$,

- s_i^0 still mediates (x_i, y_i) and
- s_i^1 still mediates (y_i, x_{i+1}).

Then the run

$$r := t_0^0 s_1^0 s_1^1 s_2^0 s_2^1 \cdots s_k^0 s_k^1 t_k^1$$

is an interleaving of r_0 and r_1 (i.e. $r \in r_1 \otimes r_2$). On the other hand, $r \vdash \tau$ because $s_1^0 s_1^1 s_2^0 s_2^1 \cdots s_k^0 s_k^1$ mediates the dependence (u, v) and items b) and c) above give the transparency properties. $\Box$

Note that the proof relies on Lemma 4.40. Like that lemma, Lemma 4.42 fails to hold if assignments execute atomically as illustrated by the following example.

Example 4.43 *Consider the two programs $\pi_1 = (y := x)$ and $\pi_2 = (x := 0\,; y := 0\,; z := y)$ and the three dependence traces $\tau_1 = (1, \langle (x, y)\rangle, 1)$, $\tau_2 = (1, \langle (y, z)\rangle, 1)$, and $\tau = (1, \langle (x, z)\rangle, 1)$.*

If assignments execute atomically, π_1 has only the run $r_1 = \langle y := x\rangle$ and π_2 has only the run $r_2 = \langle x := 0, y := 0, z := y\rangle$. Clearly, τ_1 is a dependence trace of r_1 and τ_2 is a dependence trace of r_2, independently of whether assignments execute atomically or not. Moreover, $C(\tau_1, \tau_2, \tau)$ holds.

But only the following four runs are possible interleavings of r_1 and r_2:

1) $\langle x := 0, y := 0, z := y, y := x\rangle$,

2) $\langle x := 0, y := 0, y := x, z := y\rangle$,

3) $\langle x := 0, y := x, y := 0, z := y\rangle$, *and*

4) $\langle y := x, x := 0, y := 0, z := y\rangle$.

It is not hard to see that τ is compatible with none of these runs.

If, on the other hand, assignments do not execute atomically, there are also runs like

5) $\langle v := x, x := 0, y := 0, y := v, u := y, z := u\rangle$,

where u, v are virtual variables, which possess τ as a dependence trace. $\Box$

4.11.5 Proof of Theorem 4.35

We can now put the pieces together and prove Theorem 4.35. By unfolding the definitions, we have

$$\begin{aligned} \alpha(R \otimes S) &= (T_{R\otimes S}, D_{R\otimes S}) \text{ and} \\ \alpha(R) \otimes^{\#} \alpha(S) &= (T_R \cap T_S, D), \text{ where} \\ D &= \{\tau \in \mathsf{DTS} \mid \exists \tau_R \in D_R, \tau_S \in D_S : C(\tau_R, \tau_S, \tau)\}^{\dagger}. \end{aligned}$$

Consequently, we have to show $T_{R\otimes S} = T_R \cap T_S$ and $D_{R\otimes S} = D$.

"$T_{R\otimes S} \subseteq T_R \cap T_S$": If $x \in T_{R\otimes S}$, then there is a run $t \in R \otimes S$ that is transparent for x. By definition, t is an interleaving of runs $r \in R$ and $s \in S$. These runs r, s must then also be transparent for x. Thus, $x \in T_R \cap T_S$.

"$T_{R\otimes S} \supseteq T_R \cap T_S$": If $x \in T_R \cap T_S$, then there are runs $r \in R$ and $s \in S$ that are transparent for x. By bounded renaming of virtual variables these runs can be chosen such that they do not share virtual variables. Then all interleavings of these two runs are in $S \otimes R$, and all of them are transparent for x. Thus, $x \in T_{R\otimes S}$.

"$D_{R\otimes S} \sqsubseteq D$": In order to show this relationship, assume that we are given $\tau \in D_{R\otimes S}$. Then we have, by the definition of $D_{R\otimes S}$ and Lemma 4.12(1.):

$$
\begin{array}{ll}
 & \exists t \in R \otimes S : t \vdash \tau \\
\text{iff} & [\text{Definition } R \otimes S] \\
 & \exists r \in R, s \in S, t \in r \otimes s : t \vdash \tau \\
\Rightarrow & [\text{Lemma 4.36}] \\
 & \exists r \in R, s \in S, \tau_r, \tau_s \in \mathsf{DT} : r \vdash \tau_r \;\wedge\; s \vdash \tau_s \;\wedge\; C(\tau_r, \tau_s, \tau) \\
\Rightarrow & [\text{Shunting, set comprehension}] \\
 & \exists \tau_r \in \{\tau \mid \exists r \in R : r \vdash \tau\}, \tau_s \in \{\tau \mid \exists s \in S : s \vdash \tau\} : C(\tau_r, \tau_s, \tau)\,. \\
\Rightarrow & [\text{Lemma 4.39}] \\
 & \exists \tau_r \in D_R, \tau_s \in D_S, \tau' \in \mathsf{DT} : C(\tau_r, \tau_s, \tau') \;\wedge\; \tau \sqsubseteq \tau' \\
\text{iff} & [\text{Set comprehension, see below}] \\
 & \exists \tau' \in \{\tau \in \mathsf{DTS} \mid \exists \tau_R \in D_R, \tau_S \in D_S : C(\tau_R, \tau_S, \tau)\} : \tau \sqsubseteq \tau' \\
\Rightarrow & [\text{Definition } D, \text{ Lemma 4.11}] \\
 & \exists \tau' \in D : \tau \sqsubseteq \tau'\,.
\end{array}
$$

There is a little snag in the step marked by 'see below': for the direction $\Rightarrow$, we must prove that τ' can be chosen as a *short* dependence trace, which is *not* true for this step in isolation. But, it is true under the assumption that $\tau \in D_{R\otimes S}$ which underlies the whole calculation: as a consequence of this assumption τ is *short* and this implies that any τ' with $\tau \sqsubseteq \tau'$ must also be short (Lemma 4.20). A calculation, in which this step is valid in isolation, requires to furnish each of the preceeding predicates with the conjunct $\tau \in D_{R\otimes S}$, which would clutter the calculation. This is the reason why we resort to this explanation.

"$D_{R\otimes S} \sqsupseteq D$": This is shown by the following chain of implications:

$$
\begin{array}{ll}
 & \tau \in D \\
\Rightarrow & [\text{Definition of } D, \text{ Lemma 4.12(1.)}] \\
 & \exists \tau_R \in D_R, \tau_S \in D_S : C(\tau_R, \tau_S, \tau)
\end{array}
$$

$\Rightarrow$ [Definition D_R, D_S, Lemma 4.12(1.)]

$\exists r \in R, s \in S, \tau_R, \tau_S : r \vdash \tau_R \wedge s \vdash \tau_S \wedge C(\tau_R, \tau_S, \tau)$

iff [By bounded renaming of virtual variables in s]

$\exists r \in R, s \in S, \tau_R, \tau_S :$

$r \vdash \tau_R \wedge s \vdash \tau_S \wedge C(\tau_R, \tau_S, \tau) \wedge \mathsf{virtual}(r) \cap \mathsf{virtual}(s) = \emptyset$

$\Rightarrow$ [Lemma 4.42, definition $R \otimes S$]

$\exists t \in R \otimes S : t \vdash \tau$

iff [Set comprehension]

$\tau \in \{\tau \in \mathsf{DT} \mid \exists r \in R \otimes S : r \vdash \tau\}$

$\Rightarrow$ [Lemma 4.11, definition $D_{R \otimes S}$]

$\exists \tau' \in D_{R \otimes S} : \tau \sqsubseteq \tau'$.

This ends the proof of Theorem 4.35. $\square$

4.12 Base Edges

In Chapter 3 we discussed that the atomicity assumptions about assignments may vary and that this would give rise to different definitions of the non-atomic run sets $[\![x := e]\!]$ assigned to an assignment statement $x := e$. Fortunately, all reasonable choices give rise to the same abstraction which is given by the following definition:

$$[\![x := e]\!]^{\#} \;=\; (X \setminus \{x\}, \{(\iota, \langle (y, x) \rangle, \kappa) \mid \iota, \kappa \in \mathbb{B},\ y \text{ appears in } e\}) \,.$$

Whatever atomicity assumption we are working with, all runs in $[\![x := e]\!]$ will contain certain auxiliary assignments to virtual variables and a single assignment to x. No program variable except x will ever be the target of an assignment in a run in $[\![x := e]\!]$. Hence all non-atomic runs are transparent just for the program variables in $X \setminus \{x\}$, which explains the adequacy of the first component of $[\![x := e]\!]^{\#}$.

The fact that any non-atomic run contains just a single assignment to x and no other assignments to program variables implies that no dependence trace of a non-atomic run can embody a dependence sequence that is longer than one or has a destination variable different from x. Each reasonable non-atomic run induces the same dependences between program variables as $x := e$, hence the induced dependences are (y, x) where y is a variable appearing in e. Moreover, no reasonable run kills a variable in e before it reads it or kills x after it has written it, which implies that the transparency bits can be chosen arbitrarily.

All dependence traces included in the second component of $[\![x := e]\!]^{\#}$ are trivially short and $\sqsubseteq$-maximal, which implies well-definedness.

Proposition 4.44 *Suppose $x := e \in \mathsf{Stmt}$. Then $\alpha([\![x := e]\!]) = [\![x := e]\!]^\#$.* $\square$

Statement ***skip*** has just the single run ε, which is obviously transparent for all variables and has just the dependence trace $(0, \varepsilon, 0)$. Hence, we define $[\![\boldsymbol{skip}]\!]^\# = (X, \{(0, \varepsilon, 0)\})$.

Proposition 4.45 $\alpha([\![\boldsymbol{skip}]\!]) = [\![\boldsymbol{skip}]\!]^\#$. $\square$

We define the abstract interpretation of a base edge e of the underlying flow graph as the interpretation of the statement $A(e)$ associated with e: $[\![e]\!]^\# = [\![A(e)]\!]^\#$.

Proposition 4.46 $\alpha([\![e]\!]) = [\![e]\!]^\#$ *for all base edges e.* $\square$

4.13 Run-Time

The goal of this section is to show that we can compute the abstract operations $pre^\#$, $post^\#$, $;^\#$, and $\otimes^\#$ in time $2^{p(|X|)}$, where $p(x)$ is a polynomial. We emphasize that we do *neither* intend to develop 'efficient' implementations of the operations *nor* to present a very precise analysis. The results of this section will mainly be used in order to establish the 'qualitative' complexity statement that the algorithms developed later run in exponential time. We are, however, interested in uncovering the parameter of exponential growth: it is the number of program variables $|X|$ rather than the size of the parallel flow graph.

Let us investigate the most expensive operation, interleaving, to some detail. First of all, we recall its definition from Section 4.11:

$$\begin{aligned}(T, D) \otimes^\# (T', D') &= (T \cap T', D''^{\uparrow}) \text{ where} \\ D'' &= \{\tau'' \in \mathsf{DTS} \mid \exists \tau \in D, \tau' \in D' : C(\tau, \tau', \tau'')\}.\end{aligned}$$

The sets T and T' are subsets of X, the set of program variables. Computing the intersection of T and T' is cheap: if we represent these sets as bit-strings (of length $|X|$), we can clearly calculate the intersection in time $O(|X|)$ by looking through the bit-strings for T and T' once.

D and D' are antichains of short dependence traces, hence $D, D' \subseteq \mathsf{DTS}$. By Lemma 4.18, the cardinality of DTS and hence of D and D' is $O(|X|^{2|X|+2})$. This clearly is $O(2^{p_0(|X|)})$ for some polynomial $p_0(x)$ because $x^{2x+2} = 2^{\log_2(x)(2x+2)} \leq 2^{2x^2+2x}$. We can hence consider at most $O(2^{p_0(|X|)} \cdot 2^{p_0(|X|)}) = O(2^{2p_0(|X|)})$ pairs of dependence traces τ and τ' when computing D''. For each fixed pair of dependence traces τ, τ' all dependence traces τ'' with $C(\tau, \tau', \tau'')$ can be determined in time $O(2^{p_1(|X|)})$ for some polynomial $p_1(x)$. We leave it to the reader to invent some procedure for this task that realizes this rather brutal bound. Even a very naive procedure that lists all short dependence traces τ'' and then checks for each listed dependence trace whether $C(\tau, \tau', \tau'')$ holds will do. The observation that τ, τ', and τ'' are short, and hence the length of their dependence sequences is bounded by $|X| + 1$ is helpful. As a consequence, we can calculate D'' in time $O(2^{2p_0(|X|)} \cdot 2^{p_1(|X|)}) = O(2^{2p_0(|X|)+p_1(|X|)})$. Again $O(2^{p_0(|X|)})$ is an asymptotic bound for the size of D'' because $D'' \subseteq \mathsf{DTS}$. It is, therefore, not hard to see that $D''^{\uparrow}$, the second component of $(T, D) \otimes^{\#} (T', D')$, can be computed from D'' in time $O(2^{p_2(|X|)})$ for some polynomial $p_2(x)$. Hence the overall cost of computing $(T, D) \otimes^{\#} (T', D')$ is $O(|X| + 2^{2p_0(|X|)+p_1(|X|)} \cdot 2^{p_2(|X|)}) = O(2^{p(|X|)})$ for some polynomial $p(x)$.

By similar considerations we can show that the other operations can be computed in time $O(2^{p(|X|)})$ too.

Lemma 4.47 *The operations pre$^{\#}$, post$^{\#}$, ;$^{\#}$, and $\otimes^{\#}$ can be computed in time $O(2^{p(|X|)})$ for some polynomial $p(x)$.* $\square$

4.14 Conclusion

In this chapter, we have defined an abstraction of sets of non-atomic runs from which the mediated dependences can be derived. Run sets are abstracted to antichains of short dependence traces that capture the potential to mediate dependences in cooperation with a parallel environment. The abstraction also records the set of program variables for which a transparent run exists in the abstracted run set. This information is necessary to propagate the transparency bits of the dependence traces properly in sequential contexts.

We have defined abstract interpretations of the operations and constants used in the constraint systems of Section 2.5 and have shown that they precisely abstract the corresponding operations on sets of non-atomic runs. We can thus effectively determine the dependences mediated by the sets of runs characterized by the constraint systems of Section 2.5 by solving these constraint systems over the new abstract interpretation. This can be done effectively because the lattice underlying the abstract interpretation is finite. In particular, we can determine the dependences mediated by bridging runs in procedural parallel flow graphs. This information can in turn be used to completely detect copy constants and eliminate faint code, which is explained in detail in the next chapter.

In summary, the dependence traces abstraction provides us with a means to perform precise interprocedural dependence analysis in parallel programs.

Chapter 5

Detecting Copy Constants and Eliminating Faint Code

In this chapter we show that we can detect copy constants and eliminate faint code in parallel flow graphs completely relative to the non-atomic semantics. The basic idea is to evaluate the constraint system for bridging runs over the abstract domain AD from the previous section and to exploit this information.

We have seen that the abstract counterparts of the operators and constants appearing in the constraint systems in Chapter 2 precisely abstract the corresponding operators on non-atomic run sets. Moreover, the abstraction mapping $\alpha : \mathsf{NR} \to \mathsf{AD}$ is universally disjunctive (Proposition 4.26). This implies that the least solution of the constraint systems over domain AD consists just of the abstractions of the least solution over domain NR. This is commonly known in the area of abstract interpretation [7, 8] and follows directly from the following fixpoint-theoretic lemma known as Transfer-Lemma [2] or μ-Fusion Rule [23].

Lemma 5.1 (Transfer lemma) *Suppose L, L' are complete lattices, $f : L \to L$ and $g : L' \to L'$ are monotonic functions and $h : L \to L'$ (Fig. 5.1).*

If h is universally disjunctive and $h \circ f = g \circ h$ then $h(\mu f) = \mu g$, where μf and μg are the least fixpoints of f and g, respectively. $\square$

The least solution of a constraint system over some domain corresponds in a straightforward and well-known way to the least fixpoint of a function derived from the constraints. The facts recalled above ensure that the premises of the Transfer-Lemma hold for the functions f and g derived from the concrete and abstract interpretation of the constraint systems over non-atomic runs and over AD, respectively, and the transfer function h that component-wise maps the concrete interpretation x of each variable X of the constraint system to its abstraction $\alpha(x)$. Hence, by solving the constraint system for bridging runs over domain AD, we can determine the abstractions of the non-atomic bridging runs precisely. From the abstract values we can read off in particular all the dependences mediated by bridging runs: if $(T, D) \in \mathsf{AD}$ is the precise abstraction of a set R of

$$\begin{tikzcd} L \arrow[loop left, "f"] \arrow[r, "h"] & L' \arrow[loop right, "g"] \end{tikzcd}$$

Figure 5.1: The situation in the transfer lemma

(non-atomic) runs, i.e., if $\alpha(R) = (T, D)$, then (x, y) is a dependence mediated by a run in R if and only if $(1, \langle(x, y)\rangle, 1) \in D$ (Proposition 4.5).

Based on this information we can detect copy constants and eliminate faint code. Corresponding algorithms are developed in this chapter. These algorithms are not efficient: they run in exponential time. More precise statements about the dependence of the run time from the different input parameters can be found in Theorem 5.6. The point here is *not* to develop efficient algorithms—we will see in the next chapter that all these problems are intractable already for loop-free parallel programs—the point is that these problems are effectively solvable at all! This comes as a surprise, because the corresponding problems are uncomputable, if we assume atomic execution of assignments [26].

Without further ado, we present, in the remainder of this chapter, the algorithms for detection of copy constants (Section 5.1) and faint code elimination (Section 5.2). While we do not perform formal correctness proofs for these algorithms, we argue (hopefully convincingly) that the presented algorithms solve the respective problems. In our opinion a more formal argumentation would obscure rather than clarify matters here. After the presentation of the algorithms, we analyze their asymptotic run-time in Section 5.3 and finish the chapter with some concluding remarks. Throughout this chapter we assume that execution of base statements is non-atomic.

5.1 Copy Constant Detection

Roughly speaking, a variable x is a copy-constant either if it is assigned a constant value (e.g., through $x := 42$) or if it is assigned the value of another copy constant (e.g., in $y := 42\,; x := y$). Thus, in copy constant detection only assignments of the simple form $x := k$, where k is a constant or variable are interpreted, all other forms of assignments (e.g. $x := y + 1$) are (conservatively) assumed to make x non-constant [32].

Algorithm 5.1 in Fig. 5.2 reads a parallel flow graph, a program point $v \in N$, and a program variable $y \in X$ and decides whether y is a copy constant at v or not. For this purpose it first computes (in Steps 1 and 2) for each program point w the set

$$I[w] = \{x \mid e_{Main} \Longrightarrow c_w \stackrel{r}{\Longrightarrow} c_v, At_w(c_w), At_v(c_v), \hat{r} \text{ mediates dep. } (x, y)\}\,.$$

Intuitively, $I[w]$ is the set of variables that can influence the value of y at v when

Algorithm 5.1

Input: A parallel flow graph as defined in Chapter 2, a program point $v \in N$ and a program variable $y \in X$.

Output: "yes" if y is a copy constant at v; "no" otherwise.

Method:

1) Compute—by standard fixpoint iteration—the least solution over domain $(\mathsf{AD}, \sqsubseteq)$ of the constraint system for bridging runs to program point v; this gives us a value $\mathsf{B}_v^\#[u]$ for each program point u.
2) Set $I[w] := \{x \mid (1, \langle (x, y) \rangle, 1) \in \mathsf{B}_v^\#[w].2\}$ for each program point $w \in N$.
3) Set $\mathit{flag} := \mathsf{false}$ and $\mathit{val} := \mathsf{unset}$.
4) If $I[e_{Main}] \neq \emptyset$ then $\mathit{flag} := \mathsf{true}$.
5) For all base edges $e = (u, w)$ annotated by an assignment statement $x := e$ with $x \in I[w]$:
   5.1) If e is a composite expression then $\mathit{flag} := \mathsf{true}$;
   5.2) If e is a constant expression then
   if $\mathit{val} = \mathsf{unset}$ then $\mathit{val} := e$ else if $\mathit{val} \neq e$ then $\mathit{flag} := \mathsf{true}$.
6) If flag then output "no" else output "yes".

Figure 5.2: An algorithm that detects copy constants in parallel programs.

some computation is at w. Clearly, in $I[w]$ dependences of bridging runs from w to v are considered. By solving the constraint system for bridging runs from Chapter 2 over the domain $(\mathsf{AD}, \sqsubseteq)$ (Step 1), we can compute the dependence traces of bridging runs; they are given by the second component of the value $\mathsf{B}_v^\#[w]$ that is computed. From the dependence traces we can read off the dependences by Proposition 4.5 and hence determine $I[w]$ (Step 2).

The rest of the algorithm is based on the following observation: variable y is *not* a copy constant at v if and only if one of the following is true:

a) there is a variable x the initialization value of which can influence y at v;

b) there is a base edge $e = (u, w)$ annotated by an assignment $x := e$ with a composite expression e on the right hand side, and x's value at w can influence y's value at v;

c) there are two distinct base edges $e = (u, w)$ and $e' = (u', w')$ each of them annotated by a constant assignment $x := k$ and $x' := k'$, respectively, such that both x at w and x' at w' can influence y at v and $k \neq k'$.

In Step 3-6 we check whether one of these conditions is true. We use a Boolean variable *flag* that is initialized to false and is set to true once we encounter a reason for y not being a copy constant at v. Step 4 tests whether condition a) holds true. Step 5 is concerned with conditions b) and c). Each base edge is examined in turn. Step 5.1 tests whether b) holds. In order to check c), we memorize in a variable *val* the value of the constant assignment that can influence y at w encountered first. In order to check c) we simply compare the value of constant assignments encountered later with the value memorized in *val*. Variable *val* is initialized with a special value `unset` that indicates that we have not seen a constant assignment so far. Finally, Step 6 outputs the answer.

Of course we could stop the algorithm immediately, once the flag is set to true. Moreover, we can output the value stored in *val* as additional information, if we have identified y as a copy constant at v. It is the value guaranteed for y at v. It may happen that *val* has still the value `unset`; this indicates that v is an unreachable program point.

We conclude:

Theorem 5.2 *Algorithm 5.1 solves the interprocedural copy constant detection problem in parallel flow graphs relative to non-atomic interpretation of base statements.*

5.2 Faint Code Elimination

A variable x is *live* at a program point p if there is a run from p to the end of the program on which x is used before it is overwritten. By referring to [12], Horwitz et. al. [16] define a variable x as *truly live* at a program point p if there is a run from p to the end of the program on which x is used in a truly live context before being defined, where a truly live context means: in a predicate, or in a call to a library routine, or in an expression whose value is assigned to a truly live variable. True liveness can be seen as a refinement of the ordinary liveness property. We call a use of a variable x in a predicate or call to a library routine a *relevant use* of x.

Assignments to variables that are not truly live at the program point immediately after the assignment are called *faint*. Intuitively, faint assignments can not influence any predicate in the program or call of a library routine. Thus, they cannot influence the observable behavior of the program (except of producing run-time errors) and may safely be eliminated from the program. This is called *faint code elimination*.

Algorithm 5.2

Input: A parallel flow graph π as defined in Chapter 2; a mapping $R : N \to 2^X$ that associates each program point u with the set of variables relevant at u.

Output: An updated edge annotation A_{new} of the parallel flow graph in which faint code is eliminated.

Method:

1) Initialize the new annotation of flow graph edges: $A_{\text{new}} := A$.
2) For each base edge $e \in \mathsf{Base}$: $A_{\text{new}}[e] := \boldsymbol{skip}$.
3) For each $v \in N$ with $R(v) \neq \emptyset$:
   3.1) Compute—by standard fixpoint iteration—the least solution over domain $(\mathsf{AD}, \sqsubseteq)$ of the constraint system for bridging runs to program point v; this gives us a value $\mathsf{B}_v^{\#}[u]$ for each program point u.
   3.2) Set $I[w] := \{x \mid \exists y \in R(v) : (1, \langle(x,y)\rangle, 1) \in \mathsf{B}_v^{\#}[u].2\}$ for each program point $w \in N$.
   3.3) For each base edge $e = (\_, w) \in \mathsf{Base}$ with $A[e] = (x := t)$:
   if $x \in I[w]$ then $A_{\text{new}}[e] := (x := t)$.
4) Output the new edge annotation A_{new}.

Figure 5.3: An algorithm that eliminates faint code in parallel programs.

Faint code elimination is a stronger form of the classic transformation of dead-code elimination [24]. Indeed, any assignment that is dead is also faint not vice versa. The paradigmatic example is shown in Fig. 5.4. Thus, faint code elimination in general can eliminate more code from a program.

We present now an algorithm for faint code elimination in parallel programs. Faint code elimination is based on information about the relevant uses of variables. Typically, this information is derived from the output and branching statements in the program: each output statement that refers to a variable x means that the value of x is relevant at the program point just before the output statement. Similarly, a branching statement guarded by a condition b means that all variables occurring in b are relevant at the program point just before the branching statement. As our view of the source program, a parallel flow graph, is an abstraction of the actual program in which I/O statements as well as the condi-

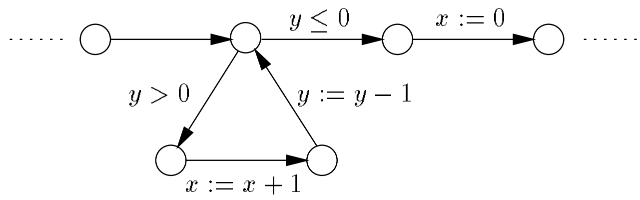


Figure 5.4: An example of an assignment that is faint but not dead. The value computed by $x := x+1$ in the loop is immediately overwritten after the loop and thus never used in a relevant context. Hence $x := x + 1$ is faint. However, it is not dead because x is potentially (non-relevantly) used by the same statement in the next iteration of the loop.

tions guarding the branching are abstracted to skip-edges, we cannot derive this information from the flow graph. Therefore, we assume that we are given this information explicitly in the form of a mapping $R : N \to 2^X$; for each program point $u \in N$, $R(u)$ is the set of variables relevant at u.

Example 5.3 *In a given source program we might find a printf statement, e.g.,*

```
printf("x+y=%d", x+y);
```

In the abstract flow graph view of the program this statement gives rise to a skip edge $e = (u, v)$. Then both `x` *and* `y` *are relevant at u, hence $R(u) = \{\mathtt{x}, \mathtt{y}\}$.*

Similarly, we might find a branching statement, e.g.,

```
if (z > x*y) then {...} else {...}
```

in the source program. In the abstract flow graph view of the program this if-statement gives rise to two skip-edges (u, v) and (u, w); u is the start node for the flow graph for the whole if-statement; at v the flow graph for the then part and at w the flow graph for the else part is found. In this case, we have $R(u) = \{\mathtt{x}, \mathtt{y}, \mathtt{z}\}$.
□

Algorithm 5.2 in Fig. 5.3 reads a parallel flow graph and a mapping $R : N \to 2^X$ as described above. Based on this information it calculates an updated version of the edge annotation mapping of the given flow graph in which faint code is eliminated. More precisely, faint instances of base statements are replaced by skip.

First the new edge annotation mapping is initialized by the original edge annotation (Step 1) and all annotations of base edges are removed, i.e. replaced by skip (Step 2). The rest of the algorithm identifies the base edges that are not faint and restores for them the original edge annotation. The algorithm is based on the simple idea that an instance of a base statement is not faint if and only if it can influence a relevant value.

We explore all program points v at which at least one variable is relevant in turn and restore the base edges that perform a computation that can influence a variable y that is relevant at v (Step 3). For this purpose we calculate in Steps 3.1 and 3.2 for all program points w the set

$$I[w] = \{x \mid e_{Main} \Longrightarrow c_w \stackrel{r}{\Longrightarrow} c_v, At_w(c_w), At_v(c_v), \exists y \in R(v) : \hat{r} \text{ med. dep. } (x, y)\}.$$

Intuitively, $I[w]$ contains the variables that can influence the value of a relevant variable y at v when some computation is at w. The computation is analogous to the one of the similar set $I[w]$ in Algorithm 5.1; therefore, we omit a detailed explanation. Step 3.3 restores the annotation of those base edges that assign to a variable that can influence a relevant variable at v from the target node of the base edge. Finally, Step 4 outputs the computed new edge annotation mapping.

We conclude:

Theorem 5.4 *Algorithm 5.2 solves the interprocedural faint code elimination problem in parallel flow graphs relative to non-atomic interpretation of base statements.*

5.3 Run-Time

The goal of this section is to analyze the asymptotic run-time of the algorithms from the previous sections. We do not intend to determine very sharp estimates but are satisfied with showing that the algorithms run in time exponential in the number of program variables, $|X|$ and polynomial in the size of the parallel flow graph. The latter is measured by the parameters $|N|$, the number of program points, $|E|$, the number of edges, and $|\mathsf{Proc}|$, the number of procedures.

In both algorithms the bulk of the work is done during the least fixpoint computation(s) for the constraint system(s) for bridging runs over the domain $(\mathsf{AD}, \sqsubseteq)$. Let us, first of all, determine an asymptotic bound for the complexity of such a fixpoint computation. As we are heading only for a rough bound, we can assume that the least fixpoint is computed naively by standard fixpoint iteration: starting from an assignment of the bottom value to each variable appearing in the constraint system we iteratively determine a new assignment to the variables by re-evaluating all constraints until convergence, i.e. until we observe no further modification. Of course the asymptotic complexity of this naive fixpoint algorithm is bounded by the product of the maximal number of iterations and the maximal cost of a single step.

In each iteration except the last one, at least one constraint variable must change its value. It is well-known that the values assigned to a constraint variable can only increase during fixpoint iteration. Therefore, for each constraint variable the value can change at most $O(|X^{2|X|+3})$ times, because $O(|X|^{2|X|+3})$ is a bound for the height of AD by Lemma 4.22. Moreover, it is a simple counting exercise to show that the complete constraint system for bridging runs (it comprises the constraint systems for same-level runs, inverse same-level runs, reaching runs, etc.) has $O(|\mathsf{Proc}|\cdot|N|)$ constraint variables.[1] Thus, we can have at most $O(|\mathsf{Proc}|\cdot|N|\cdot|X|^{2|X|+3})$ iterations. This clearly is $O(|\mathsf{Proc}|\cdot|N|\cdot 2^{p_0(|X|)})$ for some polynomial $p_0(x)$ in x, because $x^{2x+3} = 2^{\log_2(x)(2x+3)} \leq 2^{2x^2+3x}$.

Let us now bound the costs of a single iteration. In each iteration we must reevaluate all constraints. As the number of operations in a single constraint is bounded, we can get an asymptotic bound for the costs of a complete reevaluation of all constraints by multiplying a bound for the number of constraints with a bound for the maximal costs of a single operation. It is again a simple counting exercise to show that the complete constraint system for bridging runs has $O(|N|\cdot|E|)$ constraints.[2] From Lemma 4.47 we know that all operations can be computed in time $O(2^{p_1(|X|)})$ for some polynomial $p_1(x)$. Hence the cost of a single iteration is $O(|N|\cdot|E|\cdot 2^{p_1(|X|)})$.

Summarizing, the constraint system for bridging runs can be evaluated over domain $(\mathsf{AD},\sqsubseteq)$ in time $O(|\mathsf{Proc}|\cdot|N|\cdot 2^{p_0(|X|)}\cdot|N|\cdot|E|\cdot 2^{p_1(|X|)}) = O(|\mathsf{Proc}|\cdot|N|^2\cdot|E|\cdot 2^{p(|X|)})$ for $p(x) = p_0(x)+p_1(x)$. Let us fix this result for later reference as a lemma.

Lemma 5.5 *The constraint system for bridging runs can be evaluated over domain* $(\mathsf{AD},\sqsubseteq)$ *in time* $O(|\mathsf{Proc}|\cdot|N|^2\cdot|E|\cdot 2^{p(|X|)})$, *where* $p(x)$ *is a polynomial.* $\square$

Let us now turn attention to the algorithms. Clearly, in the copy constant detection algorithm, Algorithm 5.1, the bulk of the work is done in Step 1 such that the time taken for Step 1 majorizes the time taken for the other steps. Hence this algorithm runs in time $O(|\mathsf{Proc}|\cdot|N|^2\cdot|E|\cdot 2^{p(|X|)})$ by Lemma 5.5.

In the faint code elimination algorithm, Algorithm 5.2, the work performed in Step 3.1 majorizes the work done in the other steps. Step 3.1 is executed at most $|N|$ times. Consequently, Algorithm 5.2 runs in time $O(|\mathsf{Proc}|\cdot|N|^3\cdot|E|\cdot 2^{p(|X|)})$.

Clearly, only those program variables are of interest in the algorithms that appear in the parallel flow graph. We can thus assume without loss of generality, that all program variables in X appear in the parallel flow graph. As the latter constitutes part of the input to all algorithm, the input size cannot be smaller than the size of X. Obviously, the same holds for Proc, N, and E such that the size of the input clearly bounds all the parameters appearing in above run-time estimations. Hence all algorithms run in time exponential in the size of the input.

[1] This asymptotic bound holds in the special case where ASS1 and ASS2 are true as well as in the general case.

[2] Again this asymptotic bound holds for both the special and the general case.

Theorem 5.6 *Algorithms 5.1 and 5.2 run in exponential time. More precisely, Algorithm 5.1 runs in time* $O(|\mathsf{Proc}| \cdot |N|^2 \cdot |E| \cdot 2^{p(|X|)})$ *and Algorithm 5.2 in time* $O(|\mathsf{Proc}| \cdot |N|^3 \cdot |E| \cdot 2^{p(|X|)})$*.* □

Corollary 5.7 *If base statements are interpreted non-atomically, the following two problems can be solved interprocedurally in parallel flow graphs in exponential time: (1) copy constant detection and (2) faint code elimination.* □

5.4 Conclusion

We have shown in this chapter that we can detect copy constants and eliminate faint code in parallel flow graphs in exponential time, *if we do not assume that base statements execute atomically*. This should be contrasted to the result that all these problems are undecidable if assignment statements are assumed to execute atomically [26]. So, the (unrealistic) idealization from program verification "atomic execution of assignment statements" that presumably simplifies matters actually increases the difficulty of these problems from the point of view of program analysis: amazingly these problems become more tractable if we adopt a less idealized, more realistic view of execution!

These results raise the question whether there are also *efficient* algorithms for these problems. Sadly, the answer to this question is 'no', unless P=NP, as we show in the next section.

Chapter 6

Intractability

In the previous chapter, we have seen that we can detect all copy constants and eliminate faint code completely in parallel programs, if we abandon the assumption that base statements execute atomically. The presented algorithms run in exponential time, which raises the question whether there are also efficient algorithms for these problems. In this chapter we show that the answer is 'no', unless P=NP. In the conclusions of this thesis, Chapter 7, we sketch possible remedies and discuss directions of future research that may still lead to algorithms of practical interest.

We exhibit a co-NP-hardness proof by means of a reduction from the well-known SAT-problem [5, 28] that applies to both flow analysis problems. We have first presented this reduction in [25] where atomic execution of base statement has been assumed, but it remains valid if this assumption is abandoned. Unlike the reductions in [26] it only relies on active propagation along copying assignments but not on well-directed re-initialization. Interestingly, the programs constructed from SAT-instances are loop-free and do not employ procedures. Hence already the intra-procedural problems for loop-free programs are intractable. For this class of programs it is rather easy to see that the problem can also be solved in non-deterministic polynomial time: witnessing runs can visit any program point at most once and can hence be guessed even in linear time. Therefore, for this class the problems are NP-complete.

For ease of presentation we represent parallel programs in this chapter by syntactic programs rather than flow graphs.

6.1 The SAT-Reduction

An instance of SAT is a conjunction $c_1 \wedge \ldots \wedge c_k$ of *clauses* $c_1, \ldots, c_k$. Each clause is a disjunction of *literals*; a literal l is either a variable x or a negated variable $\neg x$, where x ranges over some set of variables X. We write $\bar{X} = \{\neg x_1, \ldots, \neg x_n\}$ for the set of negated variables. It is straightforward to define when a *truth* *assignment* $T : X \to \mathbb{B}$, where $\mathbb{B} = \{\mathsf{tt}, \mathsf{ff}\}$ is the set of truth values, satisfies $c_1 \wedge \ldots \wedge c_k$. The SAT problem asks us to decide for each instance $c_1 \wedge \ldots \wedge c_k$ whether there is a satisfying truth assignment or not.

From a given SAT instance $c_1 \wedge \ldots \wedge c_k$ with k clauses over n variables $X = \{x_1, \ldots, x_n\}$ we construct a loop-free parallel program. In the program we use $k+1$ variables $z_0, z_1, \ldots, z_k$. Intuitively, z_i is related to clause c_i for $1 \leq i \leq k$; z_0 is an extra variable.

For each literal $l \in X \cup \bar{X}$ we define a statement π_l. π_l consists of a sequential composition of assignments of the form $z_i := z_{i-1}$ in increasing order of i. The assignment $z_i := z_{i-1}$ is in π_l if and only if the literal l makes clause i true. Formally, $\pi_l = \pi_l^k$, where

$$\pi_l^0 \stackrel{\text{def}}{=} \textbf{\textit{skip}}$$

$$\pi_l^i \stackrel{\text{def}}{=} \begin{cases} \pi_l^{i-1}\,;\, z_i := z_{i-1}\,, & \text{if clause } c_i \text{ contains } l \\ \pi_l^{i-1}\,, & \text{if clause } c_i \text{ does not contain } l \end{cases}$$

for $i = 1, \ldots, k$. Now, consider the following program π:[1]

> **procedure** *Main*;
> $z_0 := 1\,;$
> $z_1 := 0\,; \ldots ;\, z_k := 0\,;$
> $[(\pi_{x_1} \sqcap \pi_{\neg x_1}) \parallel \cdots \parallel (\pi_{x_n} \sqcap \pi_{\neg x_n})]\,;$
> $(z_k := 0 \sqcap \textbf{\textit{skip}})\,;$
> **write**(z_k)
> **end**

Clearly, π can be constructed from the given SAT instance $c_1 \wedge \ldots \wedge c_k$ in polynomial time or logarithmic space.

It is not hard to see that the value 1 from the initialization of z_0 can be propagated to the final write statement if and only if the given SAT instance is satisfiable:

"If": On the one hand, we can construct from a satisfying truth assignment $T : X \to \mathbb{B}$ a run that propagates z_0's initialization to the write-statement. In each parallel component $\pi_{x_i} \sqcap \pi_{\neg x_i}$ we choose the left branch π_{x_i} if $T(x_i) = \mathsf{tt}$ and the right branch $\pi_{\neg x_i}$ otherwise. As T is a satisfying truth assignment, there will be, for any $i \in \{1, \ldots, k\}$, at least one assignment $z_i := z_{i-1}$ in one of the chosen branches. We interleave the branches now in such a way that the assignment(s) to z_1 are executed first, followed by the assignment(s) to z_2 etc. This results in a propagating run.

[1] The initialization of variables $z_1, \ldots, z_k$ as well as the statement $(z_k := 0 \sqcap \textbf{\textit{skip}})$ just before the write-statement is only needed for the copy constant detection problem and can be omitted for faint code elimination.

"Only if": On the other hand, a run can propagate the value with which z_0 is initialized to the write-statement only via copying it from z_0 to z_1, from z_1 to z_2 etc., because all assignments have the form $z_i := z_{i-1}$. Such a run must thus contain the assignment $z_i := z_{i-1}$ for all $i = 1, \ldots, k$. But from the way in which the non-deterministic choices are resolved in such a run we can easily construct a satisfying truth assignment.

The arguments for both directions hold independently from the atomicity assumption for assignment statements.

Consequently, the given SAT instance is satisfiable if and only if any of the following two conditions holds:

1. $z_0 := 1$ is not a faint assignment.
2. z_k is not a copy constant at the write statement.

Item 2. demands additional explanation: observe, first, that 0 and 1 are the only values z_k can hold at the write-statement because all variables are initialized by 0 or 1 and the other assignments only copy these values. Clearly, due to the non-deterministic choice just before the write-statement, z_k may hold 0 finally. Thus, z_k is a constant at the write-statement iff it cannot hold 1 there. The latter obviously holds iff the initialization value of z_0 cannot be propagated.

Example 6.1 *Fig. 6.1 shows an example clause and program for illustration. Assignments to different variables are shown on different levels. Intuitively a satisfying truth assignment corresponds to a way of resolving the non-deterministic choices in the three threads such that at each level at least one assignment is present in one of the chosen branches. This is the case if and only if the initialization may propagate to the write instruction.*

The program constructed in the above reduction is loop-free and does not employ procedures. Therefore, the reduction already applies to the intra-procedural problems for loop-free programs. For this class of programs it is rather easy to see that the problem can also be solved in non-deterministic polynomial time: Imagine a non-deterministic algorithm that guesses a run that witnesses non-constancy or non-faintness, respectively. Such a witnessing run can visit any program point at most once because the program is loop-free. Hence it can be guessed even in time linear in the program size.

These considerations prove:

Theorem 6.2 *Independently of the atomicity assumption for base statements, detecting copy constants and detecting faint code in loop-free parallel programs are co-NP-complete problems.*

Corollary 6.3 *Independently of the atomicity assumption, detecting copy constants, and detecting faint code are co-NP-hard problems in arbitrary parallel programs.*

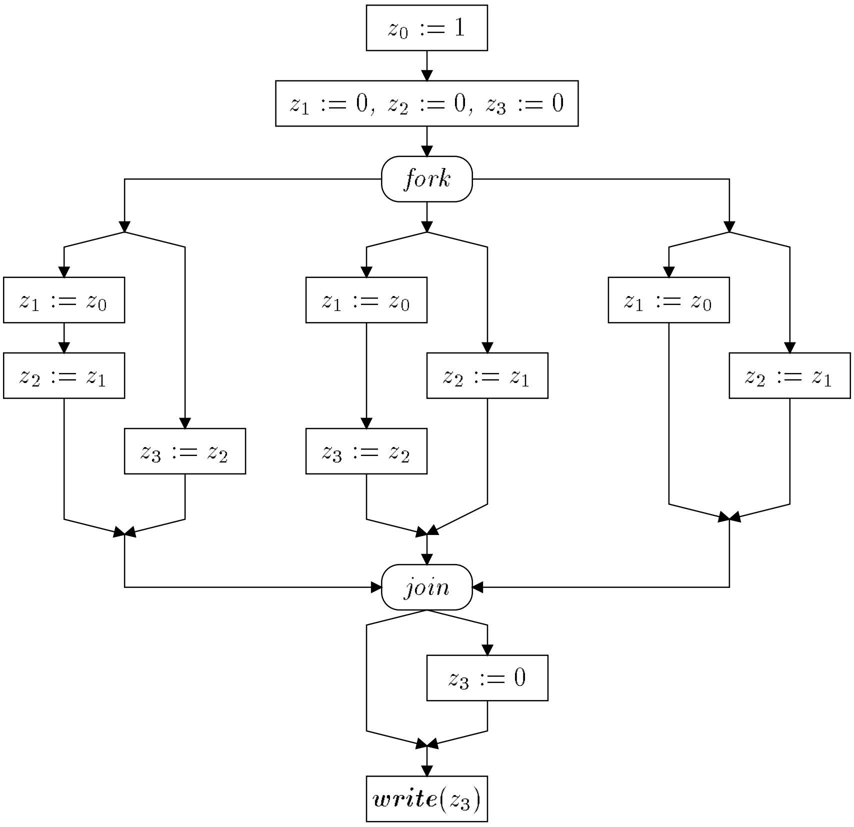


Figure 6.1: The flow graph for $(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2)$.

6.2 Summary

In this chapter, we have seen that both detecting copy constants and eliminating faint code are intractable problems, even if the assumption that base statements execute atomically is abandoned. Both problems have been shown to be co-NP-hard by means of a reduction from the SAT problem. Unlike the reductions in [26], this reduction applies under the assumption that assignments execute atomically as well as when this assumption is abandoned.

Chapter 7

Conclusion

In this report we explored the consequences for program analysis of abandoning the atomic execution assumption for base statements in parallel programs, which is the more realistic setup in practice. Surprisingly, it turned out that this enables a precise interprocedural dependence analysis of parallel programs in sharp contrast to the situation when base statement are assumed to execute atomically. Specifically, we proposed an effective abstract domain of *sets of dependence traces* which enables a precise abstract interpretation of constraint systems characterizing certain (non-atomic) run sets in parallel programs. From the values of this abstract domain the mediated dependences can be read off. We have shown how the results of the dependence analysis can be used for the interprocedural detection of copy constants and elimination of faint code in parallel programs. These algorithms solve the problems completely relative to the non-atomic semantics of parallel programs. Of course, the algorithms are sound also under the stronger execution assumption that base statements execute atomically. However, relative to atomic execution they are incomplete, which is indispensable in view of known decidability results.

The algorithms of Chapter 5 are not of direct practical relevance. Their worst-case running-time is exponential and we do not even hope that they would perform well in practice because already the elementary operations are expensive, in particular the abstract interleaving operator. Indeed, we have shown that the analysis problems are intractable (co-NP-hard). Nevertheless, we believe that refinements of the technique underlying dependence traces can lead to practically interesting algorithms that are much more precise than existing algorithms in which interference it treated rather pessimistically. Let us discuss possible targets for improvements.

1. While the run-time of the algorithms is exponential in the number of program variables, it is *polynomial in the program size*; cf. Theorem 5.6. Hence, if the number of program variables is bounded, they are polynomial-time algorithms. For a practical algorithm it is thus essential to keep the number

of variables which are used in dependence trace construction small.

In order to keep the technical treatment manageable, the current exposition does not distinguish between local and global variables of threads and procedures. Therefore, we must include all variables into the precise interference analysis provided by dependence traces. In practice, however, most variables are local to threads and there are only a few global variables on which interference can happen.

A promising direction for future research is to construct algorithms that take advantage of the distinction between local and global variables. The idea would be to devise a combined analysis that uses a cheap sequential technique for propagation via local variables and applies the expensive interference reasoning via dependence traces to global variables only. Analysis with respect to such a domain promises to be exponential only in the number of global variables, which is probably rather small in practice.

2. We can also strive for a compact representation of the values in the abstract domain AD. Each value comprises a set of variables T and an anti-chain of short dependence traces D. Set T may straightforwardly be represented by a bit-vector. It is less clear, however, what is an adequate representation for D. Storing all the dependence traces in D explicitly, e.g., as linked lists, is probably not a good solution, because D can be rather large and there is much redundancy.

   The run $\langle b := a, d := c, f := e\rangle$, for instance, has the dependence trace $\tau = (1, \langle (a,b),(c,d),(e,f)\rangle, 1)$ but also $(1, \langle (a,b),(c,d)\rangle, 1)$, $(1, \langle (a,b)\rangle, 1)$, and many others. In a certain sense the latter dependence traces are implied by τ except of the transparency bits. It is interesting to devise a representation that employs sharing to compactly represent all these dependence traces by a structure not much larger than τ alone. This representation should allow to compute the composition operators easily.

3. In flow analysis of sequential programs we mostly propagate informative values through the program. The dependence traces domain, however, is fitted to the computation of dependences only; in this sense is a rather poor and pure domain. Although it allows to solve problems like copy constant detection and faint code elimination, the approach by which this is done is rather indirect via bridging runs and involved even an iterated computation of dependence traces in the case of faint code elimination.

   It is interesting to invent and study more complex abstract domains that work with more informative values but rely on the idea of dependence traces to come to grips with interference. Ideally, such a domain should be obtained by a modular extension of the dependence traces domain in order to isolate the interference-related reasoning from other semantic questions.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] K. R. Apt and G. D. Plotkin. Countable nondeterminism and random assignment. *Journal of the ACM*, 33(4):724–767, 1986.

[3] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1990.

[4] A. Bouajjani and P. Habermehl. Constrained properties, semilinear systems, and Petri nets. In U. Montantari and V. Sassone, editors, *Concur '96*, volume 1119 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[5] S. A. Cook. The complexity of theorem-proving procedures. In *Proc. 3rd Ann. ACM Symp. on Theory of Computing*, pages 151–158, 1971.

[6] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[7] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings 4th POPL*, Los Angeles, California, January 1977.

[8] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Logic Computat.*, 4(2):511–547, 1992.

[9] J. Esparza and J. Knoop. An Automata-theoretic Approach to Interprocedural Data-flow Analysis. In *FoSSaCS '99*, volume 1578 of *Lecture Notes in Computer Science*, pages 14–30. Springer-Verlag, 1999.

[10] J. Esparza and A. Podelski. Efficient Algorithms for pre^* and $post^*$ on Interprocedural Parallel Flow Graphs. In *ACM International Conference on Principles of Programming Languages (POPL)*, pages 1–11, 2000.

[11] C. Fischer and R. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1988.

[12] R. Giegerich, U. Möncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformations. In *GI 11. Jahrestagung*, volume 50 of *Informatik Fachberichte*, pages 1–10. Springer-Verlag, 1981.

[13] D. Grunwald and H. Srinivasan. Data flow equations for explicitly parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Parallel Programming (PPOPP'93)*, 1993. SIGPLAN Notices, 28(7).

[14] C. A. Gunter and D. S. Scott. Semantic domains. In van Leeuwen [37], pages 633–674.

[15] M. S. Hecht. *Flow analysis of computer programs.* Elsevier North-Holland, 1977.

[16] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. Technical Report TR-1283, Computer Sciences Department, University of Wisconsin, Madison, WI, August 1995.

[17] inmos limited. *Transputer Instruction Set – A Compiler Writer's Guide.* Prentice Hall International, first edition, 1988.

[18] D. S. Johnson. A catalog of complexity classes. In van Leeuwen [36], pages 67–161.

[19] G. A. Kildall. A unified approach to global program optimization. In *Proceedings 1st POPL*, pages 194–206, 1973.

[20] J. Knoop. Parallel constant propagation. In D. Pritchard and J. Reeve, editors, *4th European Conference on Parallel Processing (Euro-Par)*, volume 1470 of *Lecture Notes in Computer Science*, pages 445–455. Springer-Verlag, 1998.

[21] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems*, 18(3):268–299, 1996.

[22] D. Lugiez and P. Schnoebelen. The regular viewpoint on PA-processes. *Theoretical Computer Science*, 274(1-2):89–115, 2002.

[23] Mathematics of Program Construction Group. Fixed-point calculus. *Information Processing Letters*, 53(3):131–136, 1995.

[24] S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, San Francisco, California, 1997.

[25] M. Müller-Olm. The complexity of copy constant detection in parallel programs. In A. Ferreira and H. Reichel, editors, *STACS 2001 (18th Annual Symposium on Theoretical Aspects of Computer Science)*, volume 2010 of *Lecture Notes in Computer Science*, pages 490–501. Springer, 2001.

[26] M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *STOC 2001 (33th Annual ACM Symposium on Theory of Computing)*, pages 647–656, Hersonissos, Crete, Greece, July 2001. ACM SIGACT, ACM Press.

[27] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Wiley Professional Computing. Springer-Verlag, 1999.

[28] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[29] G. D. Plotkin. A structured approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Comput. Sci. Dept., 1981.

[30] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. Technical Report RC 21493, IBM T. J. Watson Research Center, May 1999. appeared in TOPLAS.

[31] M. Rinard. Analysis of multithreaded programs. In P. Cousot, editor, *Static Analysis of Systems (SAS 2001)*, volume 2126 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, 2001.

[32] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.

[33] H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. In G. Smolka, editor, *Programming Languages and Systems (ESOP'2000)*, volume 1782 of *Lecture Notes in Computer Science*, pages 351–365. Springer-Verlag, 2000.

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

[35] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[36] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science. Volume A: Algorithms and Complexity.* Elsevier Science Publishers B.V., 1990.

[37] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science. Volume B: Formal Models and Semantics.* Elsevier Science Publishers B.V., 1990.

[38] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.