THÈSE

présentée à

l'ÉCOLE POLYTECHNIQUE

pour l'obtention du titre de

DOCTEUR DE L'ÉCOLE POLYTECHNIQUE EN INFORMATIQUE

Xavier RIVAL

21 octobre 2005

Abstraction de Traces en Analyse Statique et Transformations de Programmes

Traces Abstraction in Static Analysis and Program Transformation

École Normale Supérieure
Département d'Informatique

Cette thèse a été préparée à l'École Normale Supérieure (Paris), sous la direction de Patrick Cousot. Elle a été financée principalement par un contrat d'allocation couplée (Normalien) à l'École Polytechnique ; par ailleurs, pendant le début de cette thèse, l'auteur a également bénéficié du statut d'élève Normalien. Cette recherche a également été financée en partie par les projets Daedalus (projet européen IST-1999-20527 du programme FP5) et Astrée (projet français RNTL).

Résumé

Cette thèse est consacrée à l'étude d'abstractions d'ensemble de traces adaptées à l'analyse statique et aux transformations de programmes. Cette étude a été menée dans le cadre de l'interprétation abstraite.

Dans une première partie, nous proposons un cadre général permettant de définir des analyses effectuant un *partitionnement des traces*. Cela permet en particulier d'utiliser des propriétés définies par l'histoire des exécutions, pour écrire des disjonctions de propriétés abstraites utiles lors de l'analyse statique. Ainsi, nous obtenons des analyses plus efficaces, qui sont non seulement plus précises mais aussi plus rapides. La méthode a été implémentée et éprouvée dans l'analyseur de code C ASTRÉE, et on obtient d'excellents résultats lors de l'analyse d'applications industrielles de grande taille.

La seconde partie est consacrée au développement de méthodes permettant d'automatiser le diagnostique des alarmes produites par un analyseur tel qu'ASTRÉE. En effet, en raison de l'incomplétude de l'analyseur, une alarme peut, soit révéler une véritable erreur dans le programme, soit provenir d'une imprécision de l'analyse.
Nous proposons tout d'abord d'extraire des *slices sémantiques*, c'est à dire des sous-ensembles de traces du programmes, satisfaisant certaines conditions ; cette technique permet de mieux caractériser le contexte d'une alarme et peut aider, soit à prouver l'alarme fausse, soit à montrer un véritable contexte d'erreur. Ensuite, nous définissons des familles d'analyses de dépendances adaptées à la recherche d'origine de comportements anormaux dans un programme, afin d'aider à un diagnostique plus efficace des raisons d'une alarme.
Les résultats lors de l'implémentation d'un prototype sont encourageants.

Enfin, dans la troisième partie, nous définissons une formalisation générale de la compilation dans le cadre de l'interprétation abstraite et intégrons diverses techniques de *compilation certifiée* dans ce cadre.
Tout d'abord, nous proposons une méthode fondée sur la traduction d'invariants obtenus lors d'une analyse du code source et sur la vérification indépendante des invariants traduits.
Ensuite, nous formalisons la méthode de preuve d'équivalence, qui produit une preuve de correction de la compilation, en prouvant l'équivalence du programme compilé et du programme source.
Enfin, nous comparons ces méthodes du point de vue théorique et à l'aide de résultats expérimentaux.

Abstract

We study of abstractions for sets of traces adapted to static analysis and program transformations in the abstract interpretation framework.

In the first part, we propose a general framework for *control-based trace partitioning* in static analysis. In particular, this framework allows to use properties of the history of program executions in order to express disjunctions of abstract properties in static analyses. As a result, we obtain efficient analyses, improving not only precision but also execution time in most cases. This method was implemented in the ASTRÉE analyzer, devoted to the analysis of C programs. Moreover, we report excellent result in the analysis of large critical real world programs.

In the second part, we develop automatic techniques for the inspections of alarms produced by an analyzer such as ASTRÉE. Indeed, the analyzer is incomplete, so an alarm raised by ASTRÉE could be either a real bug or just be due to an imprecision inherent in the analysis.
First, we propose to extract *semantic slices*, i.e. subsets of the program execution traces, which satisfy some given conditions; this approach allows to characterize more precisely the context corresponding to an alarm. Furthermore, in some cases, it helps to prove the alarm to be false; otherwise, it may help to find a real error scenario. Then, we define families of dependence analyses so as to track the origin of abnormal behaviors in programs, and to help for a more efficient diagnosis of the reason why an alarm was raised.
We got encouraging results using a prototype, which we implemented.

In the last part, we define a general formalization for compilation in the abstract interpretation framework and we integrate several approaches to *certified compilation* in our framework.
First, we propose a method based on a translation of abstract invariants computed in an analysis of the source code and on the checking of the the soundness of the resulting invariants. This checking allows to trust the translated invariant independantly from any assumption about the soundness of the translation or the source analysis.
Second, we formalize the translation equivalence approach, which amounts to proving the correctness of compilation, by checking that the source program and the compiled program are equivalent.
Last, we compare both techniques not only in the theoretical point of view but also in a practical experiment.

Acknowledgments

My first thank goes to Patrick Cousot for accepting to be my advisor for my Master Thesis and then for my PhD Thesis. He gave me the chance to work on challenging topics, provided me a wonderful research environment and allowed me to be free of most important choices in my work.

I also wish to express my gratitude to my PhD Jury. First, Manuel Hermenegildo and Mooly Sagiv accepted to review it and to write reports. While doing this, they provided me a great feedback and very helpful comments. Peter Lee, Xavier Leroy, Francesco Ranzato, and Mooly Sagiv greatly contributed to the jury for my defense. I am very thankful for all the great discussions before, during and after the defense, which I could have with the jury members. In particular, the questions during the defense offered me some invaluable opportunities to improve significantly the present manuscript.

During my PhD, I had the wonderful opportunity to work on the ASTRÉE project. Taking part to this challenging embedded software certification project was a key element to the success of my work. Not only I found great topics to work on and a grand challenge to work forward, but I also could enjoy fuitful discussions with the great members of the "magic team" Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné and David Monniaux. I thank them for the wonderful, enlightning experience a research collaboration with them is. This project would never have happened without the trust of a group of engineers at Airbus: I am very thankful to Famantanantsoa Randimbilolona and Jean Souyris for submitting challenging goals to us and supporting us while they were not sure we would make it. I also wish to thank Julien Bertrane for great discussions and for his support. An additional thank goes to Julien Bertrane, Patrick Cousot, Jérôme Feret, and David Monniaux for proof reading parts of my PhD manuscript.

I am very thankful for the effort of a very efficient group of secretaries: Joelle Isnard, Valérie Mongiat, Michelle Angely and Sylvia Imbert manage the Département d'Informatique of the ENS perfectly.Moreover, Jacques Beigbeder and his assistants make a huge effort keeping up the reliability and the safety of all these computers we use so much –especially in the ASTRÉE project.

During all my effort toward this PhD, Charles Hymans, Francesco Logozzo and Yves Verhoeven always supported me.I would also like to thank all the visitors and scientists I had the chance to work, or discuss with in the ENS or at one of the many conferences I attended: Shivali Agarwal, Elie Burzstein, Guillaume Capron, Roberto Giacobazzi, George Necula, Ramona et Nicolae Mihalache, Élodie-Jane Sims...

I also need to mention all those who contributed to this thesis by giving me inspiration: Björk, the Pink Floyds, and all the friends I met while practicing photography, hiking and trekking.

Finally, I would like to thank my parents for giving me a taste for knowledge, science and research and for supporting me through all my studies.

Contents

Part I

Introduction to Traces Abstractions

Chapter 1

Introduction

1.1 Software Verification

1.1.1 Need for software verification

In the last decades, software took a growing importance into all kinds of systems. For instance, the design of the electronic command of transportation system represents typically 30 % to 40 % of the cost of the whole development; moreover, most of this fraction is due to the testing and debugging stages.

Moreover, complex systems require complex, intricate and large software. As an example, the typical size of current designs for fly-by-wire control systems ranges from 100 000 to 1 000 000 LOCs (lines of code), whereas it used to be typically 10 times smaller 10 years ago.

The consequence of this increasing complexity is that the probability for bugs and failures is dramatically augmented, unless the development process is extremely rigorous. Moreover, the consequences of a software failure range from an insignificant imprecision in the computation to the worse unexpected behavior such as the crash of a whole system; in particular, it may cause great human or economic damage, and thus, is not acceptable.

The risk for bugs to occur and to cause major damage is not theoretical. We could cite many examples of famous bugs. For instance, an integer overflow arising in a low importance task caused both the main and the backup control systems to shut down, 30 seconds after the take-off of the Ariane 501 launcher in 1996, resulting in the destruction of the rocket [ea96]. The imprecision in floating point computations caused the failure of a Patriot missile in 1992, causing dozens of deaths. Even when they do not result in a dramatic failure, bugs may cause tremendous over-costs: for example, multiple issues in the development of the baggage handling system of the Denver airport resulted in a two years schedule overrun and a $ 116 million budget overrun, in 1995. Many other "software horror stories" can be found, e.g. in `http://www.cs.tau.ac.il/~nachumd/horror.html`, ranging from the most peculiar to the most dramatic reports.

As a consequence of the importance of software, we notice an increasing interest in software verification methods.

1.1.2 Current trends in software verification

In the last thirty years a large range of software verification techniques have been developed, so as to tackle various applications.

Properties: First, let us summarize the most common properties, to be checked by software verification systems.

Safety properties express that programs "should not go wrong". In particular, the absence of runtime-errors or the absence of undefined behaviors are safety properties. Obviously, such properties are of great interest, when checking the design of critical systems, such as flight control systems. In particular, the failure of the Ariane 5 launch is the result of the violation of a simple safety property about the integer conversions. A common approach to check such properties consists in computing such an over-approximation of the real behavior of the programs, and to use this approximation in order to check that the programs never break some safety conditions.

Another family of crucial properties are **resource usage** properties. Indeed, embedded software are usually *real-time* programs, so that the overuse of memory or time resources would result in the loss of the system. **Functional** properties state that the system should perform some actions in some conditions. For instance, *liveness* properties state that a program should eventually achieve some "good condition". **Security** properties assert that non-authorized users should not be able to acquire any information about private computation or to corrupt any critical process.

In this thesis, we focus on safety properties, and we more particularly attempt at proving the absence of runtime errors. Though, most of the algorithms described in this thesis would apply to other problems.

Verification methods: The verification of software designs used to consist mainly in testing and debugging methods. The idea is to run a program with various (randomly or manually generated) sets of inputs, and to check that the properties of interest are not violated in the "test runs". However, the drawback of these solutions is that the number of possible real executions is near infinite and all situations cannot be tested. Moreover, the cost of testing is cumbersome; in particular, a change in the program should be tested exhaustively.

As a consequence, automatic, formal methods were proposed, so to increase the level of confidence in the results and to cut the cost.

The principle of **Abstract Interpretation** [CC77] based **Static Analysis** is to elaborate a model of execution for programs, then to choose an over-approximation of the program behaviors, and last, to derive analyzers for computing such over-approximation automatically. This method is *sound*, but *not complete*: in case the over-approximation satisfies all safety conditions, then the program is proved correct; otherwise, we should investigate the reasons for the failure in order to prove the safety, and conclude either that there is a real bug, or that the abstraction should be refined. In the last few years, several successful static analyzers were implemented so as to verify memory properties [LAS00], the absence of runtime errors [BCC$^+$02, BCC$^+$03a], the absence of buffer overruns [DRS03], the correctness of pointer operations [VB04].

Model Checking is a technique based on the abstraction of a program into a (e.g., boolean) model and on the application of SAT-solving methods so as to determine whether dangerous states are accessible. Modern developments in this area allowed for a refinement of the model [CGJ$^+$00] if the checking phase fails to establish the property of interest. A major difficulty of this approach is to synthesize the model from the program and the property to check; in particular the size of the model is critical for the checking phase to be practical. These techniques have been successfully applied in many areas, such as hardware verification [DSC98], software verification [BR01]...

Another approach consists in using **Theorem Proving** methods, in order to prove the correctness conditions of the program. The definition of the model is critical (if the model is wrong, the proof of correctness with respect to the model is useless), and is difficult to automatize. Furthermore, the automation of the proofs can be a major problem, whereas manual proofs incur major costs. Moreover, the adaptation of proofs for modified programs may also turn out to be very costly, compared to fully automatic methods. A major achievement of this approach was the generation of certified code following the B method [Abr89] for the most critical parts of the control system of the "Meteor" line of Paris subway, despite a high cost.

Perspectives: At this point, the use of formal methods in the development and verification of critical systems is not standard, even though we notice a growing interest in these areas.

In the last few years, various quality and safety standards have been elaborated for the most critical applications. For instance, the DO178-B regulation [TCoA99] requires the observance of strict rules in the design of software for aircrafts: the level of criticality of each part of the code should be determined, the relation between the result of successive development stages should be established, the safety of the most critical sub-systems should be ensured and verified –if possible, formally.

As a consequence, we notice an increasing need for verification tools able to tackle large critical applications. Moreover, it seems that verification methods should apply to the *real* code (the validation of a model may not be considered a sufficient guarantee). In particular, the *scalability* of the analyses is crucial, due to the growing size of the applications.

Last, it is utterly important that the method *integrate* naturally in the development process. Indeed, the verification should help in the design of better software, and not impede the development. In practice, the verification method should preferably be automatic and provide immediately usable results (readable invariants, help in the alarm investigation process).

1.1.3 Context of the the thesis

This thesis was developed in the context of the ASTRÉE project (`http://www.astree.ens.fr`). ASTRÉE [BCC$^+$03a] is an academic, abstract interpretation [CC77] based *static analyzer* developed in the École Normale Supérieure and in the École Polytechnique by Bruno BLANCHET, Patrick COUSOT, Radhia COUSOT, Jérôme FERET, Laurent MAUBORGNE, Antoine MINÉ, David MONNIAUX and myself. The ASTRÉE static analyzer aims at proving the absence of runtime errors in large, embedded programs, written in C [ANS99]; it can also be used in order to prove other classes of *safety properties*.

This project greatly impacted the choices made in this thesis, and the choice of the areas to investigate:

1. **trace partitioning**, i.e., design of trace domains, which allow to express disjunction of properties and use properties about the history of executions in order to discriminate different elements of the disjunctions;
2. **alarm investigation**, i.e., assistance to the user, when facing alarms raised by ASTRÉE, which could be either the sign of true errors or the consequence of imprecisions in the analysis;
3. **certified compilation**, so as to bring the results of analyses like ASTRÉE to the assembly level and to provide a functional certification of the compiled programs.

These three topics turn out to present *strong relations*: indeed, they all focus on **abstractions of traces**. For instance, trace partitioning is formalized as a *trace abstract domain*. The alarm investigation algorithms greatly benefit from the trace partitioning technique. Moreover, the abstractions used in the formalization of slicing and compilation are similar. As a consequence, the core of this thesis consists in the study of abstractions of sets of traces.

Last, we implemented and tested on real-world, large applications most of the algorithms and techniques presented in this Thesis.

1.2 Outline of the Thesis

In this section, we motivate, review, and summarize the main parts of the thesis.

1.2.1 Traces abstractions

In the first part, we set up our main mathematical notations and review common abstractions for sets of traces. This part should ensure the self-contained-ness of the thesis, so that a reader who is not familiar with either of the basic notions used in the following should find here the fundamental notions, whereas the knowledgeable reader can safely skip Chapter 2 and Chapter 3 and use them as a reference.

Chapter 2 sets up the syntax and the semantics of a simple imperative language, which we use throughout the rest of the thesis; it also gives a short introduction to abstract interpretation [CC77].

Chapter 3 introduces three common abstractions for sets of traces, which are widely used in the thesis:

- the static analysis based on numerical abstractions allows to derive insightful invariants about programs;
- the denotational semantics abstracts sets of traces into functions; it allows to derive efficient analyzers;
- the projection abstractions allow to focus on some observation of the history of programs.

1.2.2 Trace partitioning

The second Part is devoted to trace partitioning.

Chapter 4 sets up a framework for defining trace partitioning domains, which allow to state properties about the history of program executions and to rely on these properties in order to let disjunctions of abstract properties be handled in abstract analyses. This framework is generic; a set of partitions of the traces is taken as a parameter. Moreover, static and dynamic analyses are allowed: dynamic partitioning analyses do not fix the partitions in the beginning, which makes them rather powerful. The following two chapters instantiate this framework, so as to solve specific problems.

Chapter 5 focuses on the design and implementation of a trace partitioning domain in ASTRÉE. Basically, this domain performs a control-history based partitioning; for instance, it allows to remember what branch of a conditional statement was taken, long after the exit of the conditional (and many other similar refinements).

This domain can be seen as a generalization of [HT98], and of data-flow analyses techniques, like qualified paths-based analyses [HR80], call-string analyses [SP81]. In particular, it allows for more partitioning criteria, for more flexibility in the handling of partitions (partitions can be merged, when disjunctions are not useful anymore) and for dynamic strategies.

This technique is particularly adapted to the analysis of imperative programs, since it tends to find the properties which should guide disjunction in the control history (e.g., in tests). We provide extensive experimental data, showing the effectiveness of the approach in ASTRÉE.

Chapter 6 proposes a second instantiation of the trace partitioning framework, which is adapted to a finer analysis of the behavior of programs.

The principle is to analyze a kind of synchronous product [HLR93] of the program with an abstract system, which expresses some property about the history of executions. For instance, the abstract system may state that two properties occurred a same number of times, or that some property has just occurred for the first time. We found two application for this technique: we introduced it in order to assist the alarm investigation (Part III), but it could also be used in order to prove functional properties of programs (even though we have not deeply investigated this possibility yet).

1.2.3 Alarms diagnosis

The third Part focuses on the issue of the diagnosis of alarms raised by the ASTRÉE analyzer. Indeed, ASTRÉE is sound (it reports *all* possible errors), but *not complete*: it may fail to prove the correctness of a correct program. This is the price to pay for soundness and automation: indeed, the properties ASTRÉE attempts to prove are undecidable.

As a consequence, alarms represent a major issue for end-users. Indeed, an alarm may correspond either to an imprecision in the analysis, or to a true error. In the former case, the end-user usually expects a counter-example, so as to document the bug found by the analysis; in the latter, the user also expects some help in order to tune the parameters of the analysis or to understand the need for a new domain. The purpose of this Part is to provide the user with some assistance in this task, even if we do not propose a fully automatic solution yet (this would be a major long-term challenge).

In Chapter 7, we describe a *semantic slicing* technique, which allows to compute invariants for a subset of the traces of a program. The semantic slices are defined by abstractions, such as the data of some set of final states, some condition on the inputs of the program and some abstract system (in the sense of Chapter 6). The principle of semantic slicing is to compute abstract invariants thanks to a forward-backward static analysis.

In case the set of traces leading to the error condition of an alarm can be proved empty, then the alarm is false; otherwise, the semantic slice should help characterizing the alarm in a more precise way. Moreover, the specification of a more precise semantic slice may allow to check the occurrence of an error in some given conditions.

Early experimental results show that this techniques can be significantly helpful in the diagnosis of alarms reported by ASTRÉE. A prototype was able to prove an alarm to be false, and produced relevant semantic slices, showing several alarms real errors, in large real-world applications.

Chapter 8 introduces several notions of dependences, so as to help the alarm investigation process. Observable dependences restrict to the dependences, which can be observed in a semantic slice of a program, so that we can focus on the dependences generated by a program in a specific error context, and track the source for the error more precisely.

Abstract dependences allow for further restrictions: only dependences, which can be observed through some abstraction are retained. For instance, when looking for the cause of an overflow alarm, we suggest to look at the way large values propagate in the program, first; this way, we can find unstable retractions, or the point where values grow above some bound. We propose early experimental results as well.

1.2.4 Certification of assembly code

The fourth Part is devoted to certified compilation.

Indeed, the regulations for critical software (e.g., in aeronautics [TCoA99]) require the final code to be certified; the certification of the source code is not considered a strong guarantee, since the compiler may be buggy, and should not be trusted (if the compiler is wrong, then the compiled code may be unsafe, even though the source code is sound). We need to verify two properties: first, the compiled program should be safe (i.e., it should cause runtime errors); second it should implement the functions specified at the source level.

We formalize compilation in Chapter 9. The goal of our formalization is to state the strongest property of the source code which is retained in the compiled program, so that we can design, formalize, and compare techniques for certified compilation. Moreover, this generic framework allows for certification methods to be defined in a generic way: the algorithms of Chapter 10 and Chapter 11 are largely independent of the compiler, the optimizations and the target architecture.

In practice the correctness of the compilation of two programs boils down to the existence of a bijection between an abstraction of the semantics of the source program and an abstraction of the semantics of the compiled program. In the most simple case, this bijection can be defined by a mapping between source and assembly control states (resp. memory locations). In the case of optimizing compilation, further abstractions should be applied, which account for the loss of structure inherent in the optimizations.

We propose to translate invariants produced by a source analyzer (e.g., ASTRÉE) in Chapter 10. The idea is to use the relation between source and compiled programs, which we set up in Chapter 9 so as to derive a sound assembly invariant from a source invariant. However, the translation relies on the assumption that the compilation is correct. Therefore, we perform an independent checking of the correctness of the translated invariant: if this phase succeeds, the translated invariant can be trusted, even if the translation of the source invariant is wrong. This technique allows to prove the safety of the compiled program independently. We implemented this method and report experimental results.

We consider the equivalence checking (or translation validation) methods in Chapter 11. This technique reduces the verification of the equivalence of the source and of the compiled program to the checking of local equivalence conditions, which is the task of a theorem prover. We propose a full implementation of this technique and apply it to large programs.

Since it proves the compilation correctness, it also allows to replace the invariant checking procedure of Chapter 10, and also reduces the amount of invariants to translate. Therefore, we compare the invariant checking and the translation validation methods in the theoretical point of view and in the light of the experimental results.

Chapter 2

Semantics and Abstraction

The purpose of this chapter is to introduce the main notations to be used in the following of this thesis. We do not attempt to provide a full description of the Abstract Interpretation theory or of program semantics, so we also provide bibliographic references.

We define in Section 2.2 a syntax and semantics for a simple imperative language, which we use in the parts devoted to static analysis, slicing and certified compilation. We provide a short introduction to Abstract Interpretation in Section 2.3.

2.1 Basic mathematical notations

An order relation is a transitive, reflexive and antisymmetric binary relation; an ordering is a set together with an order relation. A *complete lattice* is an ordering $(E, <)$, such that any subset of E has a lower upper bound (lub) and a greater lower bound (glb); in particular the lub (resp. glb) of $\emptyset$ is denoted with $\bot$ (resp. $\top$); it is the least (resp. greatest) element of E. We denote lubs (resp. glbs) with $\vee$ (resp. $\wedge$).

The existence of lubs and glbs for arbitrary sets of elements is usually considered a very strong assumption; hence, we may only assume the existence of *binary* lubs and glbs. A *lattice* is an ordering with binary lubs and glbs.

If $(D, \subseteq)$ is a lattice and $F : D \to D$, then a *fixpoint* of F is an element $x \in D$ such that $F(x) = x$; a *post-fixpoint* of F is an element $x \in D$ such that $F(x) \subseteq x$. The most important results about fixpoints are:

- the set of fixpoints of a *monotone* function F over a lattice is a lattice [Tar55]; in particular, such a function enjoys a least fixpoint (denoted $\mathbf{lfp}F$) and a greatest fixpoint (denoted $\mathbf{gfp}F$). In particular, $\mathbf{lfp}F$ is the least *post-fixpoint* of F and $F(x) \subseteq x \Longrightarrow \mathbf{lfp}F \subseteq x$.
- in case F is defined over a complete lattice, and *continuous* (i.e., preserves lubs), then $\mathbf{lfp}F = \cup\{F^n(\bot) \mid n \in \mathbb{N}\}$.

Last, we write $\mathbf{Card}(E)$ for the number of elements of a set E.

2.2 Syntax and Semantics of a Simple Language

2.2.1 Syntax

We describe an imperative program with a transition system.

More precisely, we let $\mathbb{V}$ denote a set of *values*; $\mathbb{X}$ denote a finite set of *memory locations* (aka variables). A *memory state* (or *store*) describes the values stored in the memory at a precise time in the execution of the program; it is a mapping of program variables into values. A store is a function $\sigma \in \mathbb{M}$, where $\mathbb{M} = \mathbb{X} \rightarrow \mathbb{V}$.

A *control state* (or *program point*) roughly corresponds to the program counter at a precise time in the execution of the program; we usually write $\mathbb{L}$ for the set of control states.

A *state* s is a pair made of a control state $l \in \mathbb{L}$ and a memory state $\sigma \in \mathbb{M}$. We write $\mathbb{S}$ for the set of states, so $\mathbb{S} = \mathbb{L} \times \mathbb{M}$. We will consider programs may cause errors. For this purpose, we introduce an *error state*, which we denote with Ω.

A program is defined by a set $\mathbb{L}$ of control states, a set of *initial states* $\mathbb{S}^{\mathrm{i}}$, and a transition relation $(\rightarrow) \subseteq \mathbb{S} \times \mathbb{S}$, which describes how the execution of the program may step from one state to the next one.

In practice, $\mathbb{S}^{\mathrm{i}} = \{l^{\mathrm{i}}\} \times \mathbb{M}$, where $l^{\mathrm{i}} \in \mathbb{L}$ is the *entry control state*, i.e. the first point in the program.

An error may occur at state s, if $s \rightarrow \Omega$; an error occurs at state s if $s \rightarrow \Omega$ is the only transition from s. If an error may occur at state s, we say that s is a *dangerous state*. Of course, the error state shall be supposed to be blocking: $\forall s \in \mathbb{S},\ \neg(\Omega \rightarrow s)$.

We call *edge* a pair $(l_0, l_1) \in \mathbb{L}^2$, such that there exists a transition from l_0 to l_1.

2.2.2 Semantics

We assume here that a program P is defined by the data of a tuple $(\mathbb{L}, \mathbb{X}, \rightarrow, \mathbb{S}^{\mathrm{i}})$. The most common semantics for describing the behavior of transition systems is the *operational semantics*, which we sketch here. It was introduced, e.g. in [Plo81].

An execution of a program is represented with a sequence of states, called a *trace*; the semantics of the program collects all such executions:

Definition 2.2.1. Trace, Semantics.

A trace σ *is a* finite *sequence* $\langle s_0, \ldots, s_n \rangle$ *where* $s_0, \ldots, s_n \in \mathbb{S}$. *We write* $\mathbb{S}^\star$ *(or,* Σ *for short) for the set of such traces, and* **length**(σ) *for the length of* σ.
A trace of P *is a trace such that any two successive states are bound by the transition relation:* $\forall i,\ s_i \rightarrow s_{i+1}$. *The* semantics $[\![P]\!]$ *of* P *is the set of traces of* P*, i.e.* $[\![P]\!] = \{\langle s_0, \ldots, s_n \rangle \in \Sigma \mid s_0 \in \mathbb{S}^{\mathrm{i}} \wedge \forall i,\ s_i \rightarrow s_{i+1}\}$.

Note that we restrict to finite traces. Other classical definitions of operational semantics include infinite traces [Cou97a]. The infinite traces of a system correspond to non-terminating executions; they can be deduced from the finite traces.

Our choice proves sufficient for our needs in this thesis.

2.2.3 A simple language

We propose a simple instantiation for the general definitions of labeled transition systems and semantics, with a simple imperative language, which we use in the following. This language intends to modelize a small, fragment of the C language [ANS99].

Types: We consider a subset $\mathbb{T}$ of the types of the C language, including:

- float: floating point numbers [CS85];
- int: machine integers;
- bool: booleans – which are usually defined as an enumeration type in C;
- $\tau[]$: arrays of elements of type τ.

Other data types should be considered (various integers and floating point sizes, pointers, structures, enumerations types, unions).

Values: Each basic type corresponds to a set of values:

- $\mathbb{F}$ is the set of n bits IEEE-754 floating point values
- $\mathbb{I} = \{-2^m, \ldots, 2^m - 1\}$ denotes the machine integer values
- $\mathbb{B} = \{\mathbf{true}, \mathbf{false}\}$ denotes the set of booleans

Hence, the set of values is $\mathbb{V} = \mathbb{F} \uplus \mathbb{I} \uplus \mathbb{B}$, unless specified otherwise.

L-values and memory locations: An *l-value* $l \in \mathbb{l}$ is a special expression, which evaluates into a memory location(s): variables, array look-ups are l-values (See the grammar on Figure 2.1(a)).

A scalar variable (i.e. integer, boolean, floating point) corresponds to a unique memory location. A variable t of type array $\tau[]$ corresponds to a pointer to a region in the memory; an array has a length $n \in \mathbb{N}$, which denotes the size of the corresponding region. The l-value $t[i]$ stands for the i-th cell of the array t; it corresponds to the i-th *sub*-region of t.

The semantics of an l-value maps a store into a memory location (in the case where non-determinism is allowed in expression, it would return a *set of* memory locations). We do not define it formally here, since it would be straightforward, yet technical: hence, in the following, we consider the case of variables only and abusively do not distinguish a variable and the corresponding memory location.

Expressions: An expression $e \in \mathbb{e}$ is either a constant, or an l-value, or a unary operator $\ominus \in \{-, \neg, \mathbf{cast}_{\tau \to \tau'}\}$ applied to one expression, or a binary operator $\oplus \in \{+, \star, \wedge, \ldots\}$; it evaluates into a scalar type value. Note that the semantics of an expression $[\![e]\!]$ maps a store into a value; this definition rules out non-determinism and errors at the level of expressions (they will be handled at the level of statements). The syntax of expressions can be found on Figure 2.1(a). The semantics of expressions is defined by straightforward induction on the syntax:

- if $v \in \mathbb{V}$, then $[\![v]\!](\rho) = v$;
- if x is a variable, then $[\![x]\!](\rho) = \rho(x)$ (the case of general l-values would be: $[\![l]\!](\rho) = \rho([\![l]\!](\rho)))$;
- if $e \in \mathbb{e}$, then $[\![\ominus e]\!](\rho) = f_\ominus([\![e]\!](\rho))$ where $f_\ominus$ is the semantic interpretation of $\ominus$ (the case of binary expressions is similar).

In case, we consider non-determinism (e.g., if we introduce a random expression $\mathbf{rnd}(V)$, which may evaluate to any value in $V \subseteq \mathbb{V}$), then the semantics of an expression maps a store into a *set of values* ($[\![e]\!] : \mathbb{M} \to \mathcal{P}(\mathbb{V})$).

Statements: Programs are made of statements. A statement $s \in \mathbb{s}$ is either a sequence of statements $s_0; \ldots; s_n$ (also called block, denoted with b), or an assignment $x := e$ (where $x \in \mathbb{l}$, $e \in \mathbb{e}$), or a conditional $\mathbf{if}(e)\, s_0\, \mathbf{else}\, s_1$ (where e is an expression and s_0, s_1 are statements), or a loop statement $\mathbf{while}(e)\, s_0$ (where e is an expression and s_0 is a statement). Moreover, we define the two following kinds of statements, so as to model non-determinism and errors:

- The $\mathbf{input}(\ell \in V)$ statement (where $\ell \in \mathbb{l}$ and $V \subseteq \mathbb{V}$) reads a random value in V and writes it into the memory location corresponding to x.
- The $\mathbf{assert}(e)$ statement (where $e \in \mathbb{e}$) checks that the condition e holds; otherwise, it causes an error.

Control states: We defined control states in Section 2.2.2. We assign a control state to each statement, which corresponds to the status of the execution right before the statement is executed. Moreover, there is a control state right at the end of each block.

Transition relation: The rules defining the transition relation are defined on Figure 2.1(b). If $\rho \in \mathbb{M}$, $x \in \mathbb{X}$, $v \in \mathbb{V}$, we write $\rho[x \leftarrow v]$ for the store obtained by writing the value v into variable x in the store ρ; $\rho[x \leftarrow v]$ is such that $(\rho[x \leftarrow v])(x) = v$ and $y \neq x \Rightarrow (\rho[x \leftarrow v])(y) = \rho(y)$.

2.2.4 Extension with procedures

In some cases, we will consider procedures as well. Figure 2.2 displays a very rough extension of the mini-language introduced in Section 2.2 into a language with procedures. Basically, a function call statement branches to the control state at the beginning of the called function; the function return branches back to the point right after the call statement.

Of course, a function might be called during the execution of another function, so that a *stack* is required in order to recover the right calling point: the function call pushes the calling point onto the stack, whereas the function return pops the last calling point on the top of the stack and branches to this point. As a consequence, we have to extend the states with a stack (Figure 2.2(b)) and extend the transition relation as well (Figure 2.2(c)).

$$\begin{array}{rcll}
v(v \in \mathbb{V}) & ::= & n \in \mathbb{I} \mid f \in \mathbb{F} \mid b \in \mathbb{B} & \\
\ell\,(\ell \in \mathbb{I}) & ::= & x & \text{variable} \\
 & \mid & \ell\,[e] & \text{array look-up}(l \in \mathbb{I}, e \in \mathbb{e}) \\
e(e \in \mathbb{e}) & ::= & v & \text{value} \\
 & \mid & \ell & \text{l-value} \\
 & \mid & \ominus e & \text{unary expression}(\ominus \in \{-, \neg, \mathbf{cast}_{\tau \rightarrow \tau'}\}) \\
 & \mid & e \oplus e & \text{binary expression}(\oplus \in \{+, \star, \wedge, \ldots\}) \\
s(s \in \mathbb{s}) & ::= & \ell := e & \text{assignment} \\
 & \mid & \mathbf{if}(e)\, s\, \mathbf{else}\, s & \text{conditional} \\
 & \mid & \mathbf{while}(e)\, s & \text{loop} \\
 & \mid & \mathbf{input}(x \in V) & \text{reading of input, } V \subseteq \mathbb{V} \\
 & \mid & \mathbf{assert}(e) & \text{assert statement} \\
 & \mid & s; \ldots; s & \text{block}
\end{array}$$

(a) Grammar

$$\begin{array}{ll}
\text{assignment} & \ell_0 : x := e; \ell_1 \\
 & \quad (\ell_0, \rho) \rightarrow (\ell_1, \rho[x \leftarrow v]) \text{ where } v = [\![e]\!](\rho) \\
 & \\
\text{conditional} & l_0 : \mathbf{if}(e)\,\{\ell_0^t : s_t; \ell_1^t\}\, \mathbf{else}\, \{\ell_0^f : s_f; \ell_1^f\}\, \ell_1 \\
 & \quad (\ell_0, \rho) \rightarrow (\ell_0^t, \rho) \quad \text{if} \quad [\![e]\!](\rho) = \mathbf{true} \\
 & \quad (\ell_0, \rho) \rightarrow (\ell_0^f, \rho) \quad \text{if} \quad [\![e]\!](\rho) = \mathbf{false} \\
 & \quad (\ell_1^t, \rho) \rightarrow (\ell_1, \rho) \\
 & \quad (\ell_1^f, \rho) \rightarrow (\ell_1, \rho) \\
 & \\
\text{loop} & \ell_0 : \mathbf{while}(e)\,\{\ell_0^b : s_t; \ell_1^b\}\, \ell_1 \\
 & \quad (\ell_0, \rho) \rightarrow (\ell_0^b, \rho) \quad \text{if} \quad [\![e]\!](\rho) = \mathbf{true} \\
 & \quad (\ell_0, \rho) \rightarrow (\ell_1, \rho) \quad \text{if} \quad [\![e]\!](\rho) = \mathbf{false} \\
 & \quad (\ell_1^b, \rho) \rightarrow (\ell_0, \rho) \\
 & \\
\text{input} & \ell_0 : \mathbf{input}(x \in V); \ell_1 \\
 & \quad (\ell_0, \rho) \rightarrow (\ell_1, \rho[x \leftarrow v]) \text{ if } v \in V \\
 & \\
\text{assertion} & \ell_0 : \mathbf{assert}(e); \ell_1 \\
 & \quad (\ell_0, \rho) \rightarrow (\ell_1, \rho) \quad \text{if} \quad [\![e]\!](\rho) = \mathbf{true} \\
 & \quad (\ell_0, \rho) \rightarrow \Omega \qquad\; \text{if} \quad [\![e]\!](\rho) = \mathbf{false}
\end{array}$$

(b) Transition relation

Figure 2.1: A simple language

Programs containing functions with arguments and/or return values can be encoded into this extension of the language, thanks to variables; therefore, we do not introduce such features formally in the language.

2.2.5 Extensions

In practice, the analyses described in this thesis (and the tools implemented in the ASTRÉE project) focus on a large fragment of the C language (or of some assembly language in the last Part of the thesis). The choice of a restricted language was made so as to allow for a more concise presentation.

Among the other features of the language, which we consider, we can cite:

- all arithmetic data-types, including integer, floating-point and bit-fields (which mostly results in more cases to consider);
- more general data structures, including structures, enumerations, pointers, unions (though, dynamic memory allocation is currently not addressed);
- variables scopes (local, global...) and kinds (auto, static, volatile);
- initializers in variable declarations;
- non recursive functions, with parameters and/or return values;
- classical control structures, including **switch** statements, forward **goto** statements;
- library functions can be handled thanks to *stubs*, i.e. pieces of code modelizing their effect (or the observation of their effect we wish to consider), by over-approximating the possible modification of the values in the environment.

Most of the above features could be added into the simple language, which we introduced above either by adding some extra cases or by encoding new features into the simpler constructions.

The main C language features which are currently not considered are:

- recursive functions;
- dynamic memory allocation.

The reason for the choice of this fragment of the C language stems from the nature of the programs considered in the ASTRÉE project: at the time we write this thesis, we mostly considered families of critical embedded programs (more precisely described in Section 5.1.1), which should include neither recursion nor dynamic memory allocation due to specific safety constraints for real-time systems.

2.3 Abstract Interpretation

In most cases, the concrete semantics is not adequate for automatic reasoning, since it is infinite, and not decidable. In particular, the operational semantics introduced in Section 2.2.2 is not decidable. In this section, we recall the most basic results of the abstract interpretation framework [CC77, CC79], which we use in the following in order to design sound, decidable or useful, approximate semantics, in order to prove properties about

Set of function names	a finite set $\mathbb{F}$
Extension of statements	$s(\in \mathbb{s}) ::= \ldots \mid \mathbf{call}\, f$
Functions	a function is a pair $(f, s) \in \mathbb{F} \times \mathbb{s}$
Program	a program p is defined by:
	a set of functions f_p
	a main function m_p

(a) Syntax

Stacks	$\mathbb{k} = (\mathbb{F} \times \mathbb{L})^\star$ (finite sequences of function names)
States	$\mathbb{S}_{\mathrm{f}} = \mathbb{k} \times \mathbb{L} \times \mathbb{M}$

(b) States

We define a new transition relation $(\rightarrow_{\mathrm{f}}) \in \mathbb{S}_{\mathrm{f}} \times \mathbb{S}_{\mathrm{f}}$, as follows:

Call statement: $l_0 : \mathbf{call}\, f; l_1 :$

$$(\kappa, l_0, \rho) \rightarrow_{\mathrm{f}} ((l_1, f) \cdot \kappa, l, \rho)$$
$$\text{where} \begin{cases} l \text{ is the entry control state of } f \\ l_0 \text{ is the calling point} \\ l_1 \text{ is the return point} \end{cases}$$

Return:

$$((l_1, f) \cdot \kappa, l, \rho) \rightarrow_{\mathrm{f}} (\kappa, l, \rho)$$
$$\text{where} \begin{cases} l \text{ is the exit point of procedure } f \\ l_1 \text{ is the return point saved on the stack} \\ \kappa \text{ is the stack } \textit{before} \text{ the call} \end{cases}$$

Other statements: if $(l, \rho) \rightarrow (l', \rho')$ in the code of function f, then:

$$\forall \kappa \in \mathbb{k},\ l_c \in \mathbb{L},\ ((l_c, f) \cdot \kappa, l, \rho) \rightarrow_{\mathrm{f}} ((l_c, f) \cdot \kappa, l', \rho')$$

Initial states: $\mathbb{S}_{\mathrm{f}}^{\mathrm{i}} = \{(l_{\mathrm{i}}, \epsilon)\} \times \mathbb{M}$

an initial state is defined by

$$\begin{cases} \text{the empty stack } \epsilon \\ \text{the entry point } l_{\mathrm{i}} \text{ of the main function } m_p \\ \text{any memory state } \rho \in \mathbb{M} \end{cases}$$

(c) Semantics

Figure 2.2: Procedural extension of a simple language

programs and program transformations.

2.3.1 Notion of abstraction

Section 2.2.2 described a form of operational semantics, which is very convenient in order to express the meaning of programs, by completely detailing their executions. Indeed, the trace semantics fully describe the behavior of programs. However, part of the peculiar details of the operational semantics can or should generally be abstracted away in order to design static analyses and program transformation schemes. In this section, we write $(D, \subseteq)$ for the ordering underlying the concrete domain.

An abstract semantics assigns denotations to programs in an abstract domain. An *abstract domain* [CC77] is an ordering $(D^\sharp, \sqsubseteq)$, related with the concrete domain. Intuitively, an element of $D^\sharp$ can be seen as a property of programs; the ordering can be considered a precision ordering: $x^\sharp \sqsubseteq y^\sharp$ means that the property $x^\sharp$ is stronger than the property $y^\sharp$. Note that other orderings might be considered in the abstract level (decidable subset of $\sqsubseteq$, termination ordering). A comprehensive discussion of abstract interpretation frameworks can be found in [CC92b].

The correspondence between the concrete and the abstract domains is the most crucial step in the definition of an abstraction. A *soundness relation* is a set $R \subseteq D \times D^\sharp$, such that $(x, x^\sharp) \in R$ if and only if x enjoys the abstract property $x^\sharp$. In practice, tighter relations can often be exhibited between the concrete domain and the abstract domain:

- a *concretization function* $\gamma : D^\sharp \rightarrow D$ maps an abstract property $x^\sharp$ into the greatest concrete element (e.g., the largest set of traces) which enjoys property $x^\sharp$;
- an *abstraction function* $\alpha : D \rightarrow D^\sharp$ maps a concrete element x into the strongest abstract property $x^\sharp$ which holds true for x.

If they exist, these "adjoint" functions are monotone.

Obviously these functions may not always exist, as shown in the following example:

Definition 2.3.1. Non-existence of α.

We consider sets of points in the 2-dimensions plane. Abstract elements are convex polyhedra [CH78], i.e. conjunctions of constraints of the form $ax + by \leq c$, where a, b, c are real numbers. Let E be the disc $x^2 + y^2 \leq 1$. It is well-known that there is no best approximation of E in the set of polyhedra, even if one can find "arbitrarily good" approximations of the E in the domain of polyhedra, as shown on Figure 2.3.

In this thesis, we always assume the existence of a concretization function; in some cases abstraction functions will be available as well.

In favorable cases, both functions exist and form a *Galois connection* [CC77]:

Definition 2.3.1. Galois-connection.

A Galois connection *between $(D, \subseteq)$ and $(D^\sharp, \sqsubseteq)$ is a pair of function (α, γ) such that:*

$$\forall x \in D,\ \forall y \in D^\sharp,\ \alpha(x) \sqsubseteq y \iff x \subseteq \gamma(y)$$

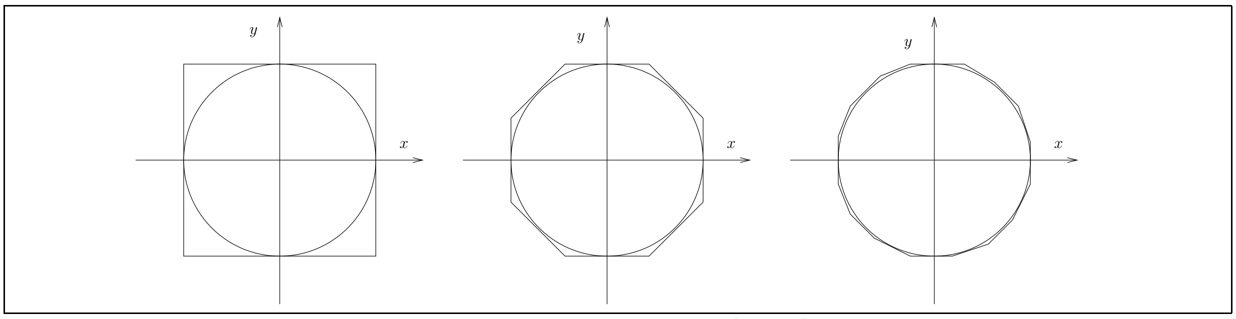


Figure 2.3: Approximations of the disc $x^2 + y^2 \leq 1$ with polyhedra

Such a Galois connection is denoted $(D, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D^\sharp, \sqsubseteq)$

For a complete overview of the many properties of Galois-connection and abstraction relations, we refer the reader to [CC92a].

2.3.2 Semantics as fixpoints and Semantic approximation

We now show how one can design a sound approximation for a concrete semantics in some given abstract domain; in practice, the construction does not depend on the abstract domain, which can be seen as a parameter.

Semantics as fixpoints: We can define the concrete semantics introduced in Section 2.2.2 as a least fixpoint, in the complete lattice $(\mathcal{P}(\Sigma), \subseteq)$:

Lemma 2.3.1. Fixpoint form for program semantics.
The semantics of P is such that:

$$[\![P]\!] = \mathbf{lfp}^{\subseteq}_{\mathcal{S}^{\mathrm{i}}} F_{\overrightarrow{P}}$$

where $F_{\overrightarrow{P}}$ is the semantic function, *defined by:*

$$\begin{array}{rccl} F_{\overrightarrow{P}}: & \Sigma & \rightarrow & \Sigma \\ & \mathcal{E} & \mapsto & \mathcal{E} \cup \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid \langle s_0, \ldots, s_n\rangle \in \mathcal{E} \wedge s_n \rightarrow s_{n+1}\} \end{array}$$

and $\mathcal{S}^{\mathrm{i}}$ collects the "initial traces", i.e. the traces made of one initial state (since any state is supposed initial in $[\![P]\!]$, so $\mathcal{S}^{\mathrm{i}} = \{\langle s\rangle \mid s \in \mathbb{S}^{\mathrm{i}}\}$).

Proof.
First, let us note that the function $F_{\overrightarrow{P}}$ is a monotone function over the lattice $(\mathcal{P}(\Sigma), \subseteq)$, so it has a least-fixpoint (as mentioned in Section 2.1). Second, $F_{\overrightarrow{P}}$ is continuous, defined on a complete lattice, so its least-fixpoint satisfies the following equality:

$$\mathbf{lfp}_{\mathcal{S}^{\mathrm{i}}} F_{\overrightarrow{P}} = \bigcup_{n \in \mathbb{N}} F_P^n(\emptyset)$$

where F_P^n denotes the n-th iterate of $F_{\overrightarrow{P}}$.
Proving that $\llbracket P \rrbracket$ is equal to the least-fixpoint amounts to proving by induction on n the property

$$\forall \sigma \in \Sigma,\ \mathbf{length}(\sigma) \leq n+1 \Longrightarrow \left(\sigma \in \llbracket P \rrbracket \iff \sigma \in \bigcup_{k=0}^{n} F^k(\mathcal{S}^{\mathrm{i}}) \right)$$

(the induction is straightforward) □

In practice most semantics can be written as least-fixpoints in a similar way.

Relations among fixpoints: The design of an abstract semantics usually follows from choice of a concrete semantics and of an abstraction by applying a "fixpoint-transfer theorem", such as:

Theorem 2.3.2. Fixpoint transfer.

We assume that $D, D^\sharp$ are complete lattices, and we let $x \in D$, $y \in D^\sharp$. Let $F : D \to D$ and $F^\sharp : D^\sharp \to D^\sharp$. Then:

- *if $\alpha : D \to D^\sharp$ is an abstraction function, $\alpha(x) = y$ and $\alpha \circ F = F^\sharp \circ \alpha$, then $\alpha(\mathbf{lfp}_x F) = \mathbf{lfp}_y F^\sharp$.*
- *if $\gamma : D^\sharp \to D$ is a concretization, $x \subseteq \gamma(y)$, and $F \circ \gamma \subseteq \gamma \circ F^\sharp$, then $\mathbf{lfp}_x F \subseteq \gamma(\mathbf{lfp}_y F^\sharp)$.*

Proof.

Such results can be proved by straightforward inductions on the sequences of iterates. □

Another noticeable fact is that a fixpoint might be checked by computing only one iterate:

Theorem 2.3.3. Fixpoint checking.

Let $F^\sharp : D^\sharp \to D^\sharp$, and $x^\sharp \in D^\sharp$. We write x for $\gamma(x^\sharp)$. Let us assume that:

- *there is a concretization function $\gamma : D^\sharp \to D$ (as usual, we assume it is monotone);*
- *the concrete semantic function $F : D \to D$ is monotone;*
- *$F^\sharp$ abstracts F, i.e., $F \circ \gamma \subseteq \gamma \circ F^\sharp$;*
- *$F^\sharp(x^\sharp) \sqsubseteq x^\sharp$*

Then, $\mathbf{lfp} F \subseteq x$.

Proof.

Since $F^\sharp$ abstracts F, $F(x) = F \circ \gamma(x^\sharp) \subseteq \gamma \circ F^\sharp(x^\sharp)$; moreover, γ is monotone, so $\gamma \circ F^\sharp(x^\sharp) \subseteq \gamma(x^\sharp) = x$; by transitivity, $F(x) \subseteq x$, so $\mathbf{lfp} F \subseteq x$. □

2.3.3 Enforcing termination

The fixpoint-transfer scheme presented in Section 2.3.2 leaves one issue to be addressed: the sequences of abstract iterates might be infinite, in case the abstract domain has infinite increasing chains. Therefore, in case we wish the abstract semantics to be computable, we replace the abstract join operator with a *widening operator* [CC77], which is an approximate join [CC92b], with additional termination properties:

Definition 2.3.2. Widening operator.

A widening *is a binary operator ∇ on $D^\sharp$, which satisfies the two following properties:*

1. $\forall x^\sharp, y^\sharp \in D^\sharp,\ x^\sharp \sqsubseteq x^\sharp \nabla y^\sharp \wedge y^\sharp \sqsubseteq x^\sharp \nabla y^\sharp$
2. *For any sequence $(x_n)_{n\in\mathbb{N}}$, the sequence $(y_n)_{n\in\mathbb{N}}$ defined below is not strictly increasing:*

$$\left\{ \begin{array}{rrcl} & y_0 & = & x_0 \\ \forall n \in \mathbb{N}, & y_{n+1} & = & y_n \nabla x_{n+1} \end{array} \right.$$

It is possible to replace property 1 with the weaker property $\gamma(x^\sharp) \cup \gamma(y^\sharp) \subseteq \gamma(x^\sharp \nabla y^\sharp)$, and recover the same properties of the widening operator.

The following theorem [CC77] shows how widening operators makes it possible to compute in a finite number of iterations a sound over-approximations for the concrete properties:

Theorem 2.3.4. Abstract iteration with widening.

We assume a concretization $\gamma : D^\sharp \to D$ is defined and that $F^\sharp$ is such that $F \circ \gamma \subseteq \gamma \circ F^\sharp$. Let $x \in D$, $x^\sharp \in D^\sharp$, such that $x \subseteq \gamma(x^\sharp)$. We define the sequence $(x_n)_{n\in\mathbb{N}}$ as follows:

$$\left\{ \begin{array}{rrcl} & x_0 & = & x^\sharp \\ \forall n \in \mathbb{N}, & x_{n+1} & = & x_n \nabla F^\sharp(x_n) \end{array} \right.$$

Then, the sequence $(x_n)_{n\in\mathbb{N}}$ is ultimately stationary and its limit $\lim(x_n)_{n\in\mathbb{N}}$ is a sound approximation of $\mathbf{lfp}_x F$:

$$\mathbf{lfp}_x F \subseteq \gamma(\lim(x_n)_{n\in\mathbb{N}})$$

Proof.

The termination follows from Definition 2.3.2, property 2 applied to the sequence $(y_n)_{n\in\mathbb{N}} = (F^\sharp(x_n))_{n\in\mathbb{N}}$.

The soundness can be proved by induction over the iterates:

$$\forall n \in \mathbb{N}, \bigcup_{k=0}^{n} F^k(\emptyset) \subseteq \gamma(x_n)$$

(this proof relies on the fact that the widening approximates concrete joins). □

Theorem 2.3.4 describes the scheme of a classical static analysis: apply a sound abstract counterpart to the concrete semantic function and widening until success of a termination test (i.e., $F^\sharp(x_n) = x_n$).

Definition 2.3.2. Non monotonicity of widening.

A classical widening on the abstract domain of intervals removes non-stable constraints. For instance, if we consider only widening on the right bound, then $[a,b]\nabla[a,c]$ *is* $[a,c]$ *if* $c \leq b$ *(stable constraint) and* $[a,+\infty[$ *otherwise (unstable constraint).*
This operator obviously enforces the convergence of any sequence of iterates after two iterations.
Let us consider the abstract function $F^\sharp : [a,b] \mapsto [a+5, \min(b+10, 50)]$. *Carrying out an abstract iteration from* $I = [0,50]$ *converges in one iteration (*$I\nabla F^\sharp(I) = I$*); the iteration starting from* $I' = [0,10]$ *requires converges after the second iteration and the limit is* $[0,+\infty[$*, which is a less precise limit even though* $I' \sqsubseteq I$*. This simple case exemplifies the non-monotonicity induced by widening operator. In practice, abstract transformers are rarely monotone.*

In practice, the result of a widening iteration can often be improved:

Remark 2.3.1. Decreasing iteration.

We keep the notations of Theorem 2.3.4 and let $x_1^\sharp$ *be the limit of the widening sequence. Since* $\mathbf{lfp}_x F \subseteq \gamma(x_1^\sharp)$ *and* $F(\mathbf{lfp}_x F) = \mathbf{lfp}_x F$*, and* γ *is monotone, we can conclude that* $\mathbf{lfp}_x F \subseteq \gamma(F^\sharp(x_1^\sharp))$*. By induction, we can show that we can apply an arbitrary number of times the operator* $F^\sharp$*, and still get a sound over-approximation of the concrete least fixpoint.*
In practice, such a sequence may noticeably improve the precision. The termination of this "post-widening" sequence is usually enforced with a narrowing *operator [CC77].*

Last, we point out that other, more general definitions for widening operators might be used in practice; in particular, the termination assumption may be asserted for a different ordering than the precision ordering (See [CC92b] for more details).

2.3.4 Program transformations

We shall also use the notion of abstraction in order to compare the semantics of programs resulting from program transformations. Basically, a program transformation is a function $\mathfrak{F}$ mapping a program into another program.

Semantic abstraction allows to describe the transformation in the semantic level. This approach was suggested by [CC02].

Let D_s (resp. D_t) be the concrete domain for expressing the semantics of source (resp. target) programs. We assume two abstractions $(D_s^\sharp, \alpha_s, \gamma_s)$ and $(D_t^\sharp, \alpha_t, \gamma_t)$, of D_s and D_t

respectively, can be defined such that there exists a function $\llbracket\mathfrak{F}\rrbracket : D_s^\sharp \rightarrow D_t^\sharp$ such that, for all program p, then $\alpha_t(\llbracket\mathfrak{F}(p)\rrbracket) = \llbracket\mathfrak{F}\rrbracket(\alpha_s(p))$. Then, we say that $\llbracket\mathfrak{F}\rrbracket$ provides a semantic definition for the program transformation as shown on the diagram below.

$$\begin{tikzcd}
P \arrow[r, mapsto, "\mathfrak{F}"] \arrow[d, mapsto, "\text{semantics}"'] & \mathfrak{F}(P) \arrow[d, mapsto, "\text{semantics}"] \\
\llbracket P\rrbracket \arrow[d, "\alpha_s"] & \llbracket\mathfrak{F}(P)\rrbracket \arrow[d, "\alpha_t"] \\
\alpha_s(\llbracket P\rrbracket) \arrow[r, mapsto, "\llbracket\mathfrak{F}\rrbracket"'] & \alpha_t(\llbracket\mathfrak{F}(P)\rrbracket)
\end{tikzcd}$$

In particular, in case both semantics are defined using least-fixpoints, then we expect $\llbracket\mathfrak{F}\rrbracket$ to relate the execution steps of the source and compiled programs.

Next chapter provides suitable abstractions of sets of traces, for the formalization of program transformations; in particular, an example will be provided in Section 3.4.

Chapter 3

Abstractions of Sets of Traces

This chapter is devoted to simple abstractions for sets of traces, which will be thoroughly used in the following of the thesis: Section 3.1 describes abstraction for static analysis; Section 3.2 defines denotational abstractions for sets of traces, as functions mapping states into states. Section 3.3 introduces a backward semantics. Section 3.4 deals with projection abstractions.

3.1 Static Analysis

This section introduces the structure of a simple abstract interpreter. We detail the structure and the implementation of the ASTRÉE analyzer [BCC+02, BCC+03a, CCF+05] later, in Section 5.1: ASTRÉE is quite different from the simple abstract interpreter described here. However, the abstractions introduced here will be used throughout the rest of the thesis.

3.1.1 The abstraction

Set of traces of interest: In this section, we consider a program P defined by the data of a tuple $(\mathbb{X}, \mathbb{L}, \mathbb{S}^{\mathrm{i}}, \rightarrow)$. We focus on the approximation of the *executions* of P, i.e. on the states which appear in a trace of P. As a consequence, we wish to approximate the set of traces $\mathcal{T} = \{\langle s_0, \ldots, s_n\rangle \mid \exists \rho_0, s_0 = (l^{\mathrm{i}}, \rho_0)\}$. We recall that that $\mathcal{T} = \mathbf{lfp}_{\mathbb{S}^{\mathrm{i}}} F$.

We proceed to the abstraction of traces into *reachable states*: we wish to abstract the traces into an approximation for the set of states $\mathcal{S}$ which appear in at least one tract in $\mathcal{T}$. In the following, we approximate all the states distinct from Ω: deciding whether Ω is reachable from the set of all reachable, non-error states is usually straightforward (it amounts to checking whether there exists a state s such that $s \rightarrow \Omega$ in the set $\mathcal{S}$).

Abstraction of traces: We assume that an abstract domain $(D^\sharp_{\mathbb{M}}, \sqsubseteq)$ for representing sets of stores is defined, together with a concretization function $\gamma_{\mathbb{M}} : D^\sharp_{\mathbb{M}} \rightarrow \mathcal{P}(\mathbb{M})$.

We let the abstraction for approximating the concrete semantics be defined by:

- the abstract domain $D^\sharp = \mathbb{L} \rightarrow D^\sharp_{\mathbb{M}}$, with the pointwise ordering induced by $\sqsubseteq$ (which we also write $\sqsubseteq$);
- the concretization function $\gamma : \imath \in D^\sharp \mapsto \{\langle(l_0, \rho_0), \ldots, (l_n, \rho_n)\rangle \mid \forall i,\ \rho_i \in \gamma_{\mathbb{M}}(\imath(l_i))$.

Intuitively, this very simple abstraction collects the memory states corresponding to each control state and applies the store abstraction to the resulting sets of stores.

Abstract operations: Moreover, we assume that the domain $D^\sharp_{\mathbb{M}}$ provides some *sound abstract operations*:

- a *least* element $\bot$, such that $\gamma_{\mathbb{M}}(\bot) = \emptyset$;
- a *greatest* element $\top$, such that $\gamma_{\mathbb{M}}(\top) = \mathbb{M}$;
- an abstract join operator $\sqcup$, approximating the concrete operator $(\forall x, y \in \mathcal{P}(\mathbb{M}),\ x^\sharp, y^\sharp \in D^\sharp_{\mathbb{M}},\ x \subseteq \gamma_{\mathbb{M}}(x^\sharp) \wedge y \subseteq \gamma_{\mathbb{M}}(y^\sharp) \Longrightarrow x \cup y \subseteq \gamma_{\mathbb{M}}(x^\sharp \sqcup y^\sharp))$ and a widening operator ∇ (Section 2.3.3).
- a sound counterpart guard : $\mathbb{e} \times \mathbb{B} \times D^\sharp_{\mathbb{M}} \rightarrow D^\sharp_{\mathbb{M}}$ for the concrete testing of conditions:

$$\begin{array}{l} \forall \rho \in \mathbb{M},\ e \in \mathbb{e},\ b \in \mathbb{B},\ d \in D^\sharp_{\mathbb{M}}, \\ \quad \left.\begin{array}{rl} & \rho \in \gamma_{\mathbb{M}}(d) \\ \wedge & [\![e]\!](\rho) = b \end{array}\right\} \Longrightarrow \rho \in \gamma_{\mathbb{M}}(\mathit{guard}(e, b, d)) \end{array}$$

  Since the operator guard : $(e, b, d) \mapsto d$ trivially satisfies the above assumption, we assume that the guard operator is reductive: $\forall \rho \in \mathbb{M},\ e \in \mathbb{e},\ b \in \mathbb{B},\ \gamma_{\mathbb{M}}(\mathit{guard}(e, b, d)) \subseteq \gamma_{\mathbb{M}}(d)$.
- a sound counterpart assign : $\mathbb{l} \times \mathbb{e} \times D^\sharp_{\mathbb{M}} \rightarrow D^\sharp_{\mathbb{M}}$ for the concrete assignment:

$$\begin{array}{l} \forall \rho \in \mathbb{M},\ \forall l \in \mathbb{l},\ e \in \mathbb{e},\ d \in D^\sharp_{\mathbb{M}}, \\ \quad \left.\begin{array}{rl} & \rho \in \gamma_{\mathbb{M}}(d) \\ \wedge & [\![l]\!](\rho) = x \\ \wedge & [\![e]\!](\rho) = v \end{array}\right\} \Longrightarrow \rho[x \leftarrow v] \in \gamma_{\mathbb{M}}(\mathit{assign}(l, e, d)) \end{array}$$

- a sound counterpart forget : $\mathbb{l} \times D^\sharp_{\mathbb{M}} \rightarrow D^\sharp_{\mathbb{M}}$ for the "variable-forget" operation, which writes a random value into a variable:

$$\begin{array}{l} \forall \rho \in \mathbb{M},\ \forall l \in \mathbb{l},\ \forall v \in \mathbb{V},\ \forall d \in D^\sharp_{\mathbb{M}}, \\ \quad \left.\begin{array}{rl} & \rho \in \gamma_{\mathbb{M}}(d) \\ \wedge & [\![l]\!](\rho) = x \end{array}\right\} \Longrightarrow \rho[x \leftarrow v] \in \gamma_{\mathbb{M}}(\mathit{forget}(l, d)) \end{array}$$

Intuitively, each of these operators should mimics a common operation of the language in a *sound* (or conservative) way. For instance, the assign operation inputs a pre-condition d and an assignment and returns an over-approximation of the post-conditions, which may be reached after carrying out the assignment operation from d. Soundness is a critical requirement for the results of the analysis to be proved correct with respect to the concrete semantics; as a consequence it is considered the most important characteristic of abstract operations.

An abstract operator may be *imprecise*: for instance, assign may return an element including many *spurious* stores (for instance, it may return $\top$). Such imprecisions may result in useless invariants (e.g., it may result in a large number of false alarms, when trying to prove the safety of a program); therefore, the design of transfer functions usually attempts to avoid coarse imprecisions whenever they may affect the result of the analysis.

Section 3.1.2 defines a simple abstract interpreter for the language introduced in Section 2.2.3; Section 3.1.3 proposes ways of building instantiations for $D^\sharp_{\mathbb{M}}$.

3.1.2 Abstract interpretation of a simple semantics

First, we define a family $(\mathit{transfer}_{\ell,\ell'})_{\ell,\ell' \in \mathbb{L}}$ of sound abstract transfer functions, using the abstract operations provided in Section 3.1.1, which are displayed on Figure 3.1. It is

assignment $\ell_0 : x := e; \ell_1$

$\quad \mathit{transfer}_{\ell_0,\ell_1} : d \mapsto \mathit{assign}(x, e, d)$

conditional $\ell_0 : \mathbf{if}(e)\, \{\ell_0^t : s_t; \ell_1^t\}\, \mathbf{else}\, \{\ell_0^f : s_f; \ell_1^f\}\, \ell_1$

$\quad \mathit{transfer}_{\ell_0,\ell_0^t} : d \mapsto \mathit{guard}(e, \mathbf{true}, d)$

$\quad \mathit{transfer}_{\ell_0,\ell_0^f} : d \mapsto \mathit{guard}(e, \mathbf{false}, d)$

$\quad \mathit{transfer}_{\ell_1^t,\ell_1} = \mathit{transfer}_{\ell_1^f,\ell_1} : d \mapsto d$

loop $\ell_0 : \mathbf{while}(e)\, \{\ell_0^b : s_t; \ell_1^b\}\, \ell_1$

$\quad \mathit{transfer}_{\ell_0,\ell_0^b} : d \mapsto \mathit{guard}(e, \mathbf{true}, d)$

$\quad \mathit{transfer}_{\ell_0,\ell_1} : d \mapsto \mathit{guard}(e, \mathbf{false}, d)$

$\quad \mathit{transfer}_{\ell_1^b,\ell_0} : d \mapsto d$

input $\ell_0 : \mathbf{input}(x \in V); \ell_1$

$\quad \mathit{transfer}_{\ell_0,\ell_1} : d \mapsto \mathit{guard}((x \in V)^\sharp, \mathbf{true}, \mathit{forget}(x, d))$

where the condition $(x \in V)^\sharp$ soundly approximates $(x \in V)$:

$(\rho(x) \in V) \Longrightarrow [\![(x \in V)^\sharp]\!](\rho) = \mathbf{true}$

assertion $\ell_0 : \mathbf{assert}(e); \ell_1$

$\quad \mathit{transfer}_{\ell_0,\ell_1} : d \mapsto \mathit{guard}(e, \mathbf{true}, d)$

Figure 3.1: A simple abstract interpreter

designed so as to satisfy the following soundness property:

Lemma 3.1.1. Transfer functions soundness.
Let $\ell, \ell' \in \mathbb{L}$, $\rho, \rho' \in \mathbb{M}$, $d \in D^\sharp_{\mathbb{M}}$. *Then:*

$$\left.\begin{array}{rl} & \rho \in \gamma_{\mathbb{M}}(d) \\ \wedge & (\ell, \rho) \rightarrow (\ell', \rho') \end{array}\right\} \Longrightarrow \rho' \in \mathit{transfer}_{\ell,\ell'}(d)$$

Proof.

Straightforward case analysis. □

Furthermore, we note that the element $\mathcal{I}_0 \in D^\sharp$ defined below safely approximates the set of initial traces $\mathbb{S}^{\mathrm{i}}$ ($\mathbb{S}^{\mathrm{i}} \subseteq \gamma(\mathcal{I}_0)$):

$$\mathcal{I}_0 : \left\{ \begin{array}{lcl} l^{\mathrm{i}} & \mapsto & \top \\ l \neq l^{\mathrm{i}} & \mapsto & \bot \end{array} \right.$$

Following Theorem 2.3.2, we define the abstract interpreter as a function $F^\sharp$ defined by:

$$\begin{array}{rlcl} F^\sharp : & D^\sharp & \longrightarrow & D^\sharp \\ & \mathcal{I} & \longmapsto & \lambda(l_{\mathrm{post}} \in \mathbb{L}) \cdot \bigsqcup\{\mathit{transfer}_{l_{\mathrm{pre}}, l_{\mathrm{post}}}(I(l_{\mathrm{pre}})) \mid l_{\mathrm{pre}} \in \mathbb{L}\} \end{array}$$

This interpreter is sound:

Theorem 3.1.2. Soundness of the simple abstract interpreter.

The sequence $(\mathcal{I}_n)_{n\in\mathbb{N}}$ defined by the element $\mathcal{I}_0$ above and $\forall n \in \mathbb{N},\ \mathcal{I}_{n+1} = \mathcal{I}_n \nabla F^\sharp(\mathcal{I}_n)$ is monotone, ultimately stationary; so it has a limit $\mathcal{I}^\sharp$. Moreover, the limit $\mathcal{I}^\sharp$ is such that:

$$\mathcal{T} \subseteq \gamma(\mathcal{I}^\sharp)$$

Proof.

It follows from Lemma 3.1.1 that $F^\sharp$ is a sound approximation for the concrete semantic function F.

The result follows from Theorem 2.3.4, since $\mathcal{T} = \mathbf{lfp}_{\mathbb{S}^{\mathrm{i}}} F$. □

Note that the interpreter provided here is not particularly efficient. In particular, more care needs to be taken for the iteration strategy. Any fair strategy for applying abstract transfer functions results in a sound analysis [Cou81]; however, not all strategies are efficient:

- Applying all local transfer functions for each iteration would turn out costly and useless, since most local transfer functions would not refine any invariant, as is the case of a block of instructions with no branching. Common analyzers rely on a fair, asynchronous iteration strategy: each iteration applies *some* local transfer functions, and any local transfer functions is applied eventually. Such strategies are usually based on work-lists containing the control states a new invariant could be computed for. More details are provided on this topic in [HDT87].
- Secondly, applying the widening operator at any control state in the control flow graph would turn costly and imprecise; therefore, an adequate set of widening control states should be determined prior to the analysis, using e.g. the algorithm presented on [Bou93].

We describe the approach followed in the ASTRÉE project in Section 3.2.5. This approach follows the syntax tree of programs and only requires local invariants to be saved at loop heads (minimal invariant storage during the analysis).

Definition 3.1.1. Issues in iteration strategies.
Let us consider the transition system below:

$$\begin{tikzcd}
& (\ell_0, t) \arrow[dl] \arrow[dr] & \\
(\ell_1, t) \arrow[dr] & & (\ell_2, t) \arrow[dl] \\
& (\ell_3, t) &
\end{tikzcd}$$

Then, the strategy which computes invariant for ℓ_0, ℓ_1, ℓ_3 first, and then for ℓ_2 is not optimal: the invariant at point ℓ_3 needs to be computed twice, so the first computation of this invariant is useless.
Iteration strategies based on the program structure rather than the control flow (as in Section 3.2.5) eliminate this issue.

Backward analysis: We may want to restrict to a set of *final* states instead of a set of *initial* states as done previously. Then, we would need to implement a *backward analysis*, which is conceptually dual of a forward analysis.

Indeed, let $\mathbb{S}^{\mathrm{f}} \subseteq \mathbb{S}$ be a set of final states and T_{f} be the set of traces $\{\langle s_0, \ldots, s_n \rangle \in \Sigma \mid s_n \in \mathbb{S}^{\mathrm{f}} \wedge \forall i,\ s_i \to s_{i+1}\}$. Then, T_{f} boils down to a least fixpoint:

$$T_{\mathrm{f}} = \mathbf{lfp}_{\mathcal{S}^{\mathrm{f}}} F_{\overleftarrow{P}}$$

where:

$$\begin{array}{rlcl}
F_{\overleftarrow{P}}: & \mathcal{P}(\Sigma) & \to & \mathcal{P}(\Sigma) \\
& \mathcal{E} & \mapsto & \mathcal{E} \cup \{\langle s_{-1}, s_0, \ldots, s_n \rangle \in \Sigma \mid \langle s_0, \ldots, s_n \rangle \in \mathcal{E} \wedge s_{-1} \to s_0\}
\end{array}$$

and $\mathcal{S}^{\mathrm{f}} = \{\langle s \rangle \mid s \in \mathbb{S}^{\mathrm{f}}\}$ (see Lemma 2.3.1).

An approximation for T_{f} can be computed by abstract interpretation, by defining *backward abstract transfer functions* for each language construction and computing an abstract post-fixpoint in the same way as in Theorem 3.1.2. Backward analysis was studied, e.g. in [Cou78, Cou81].

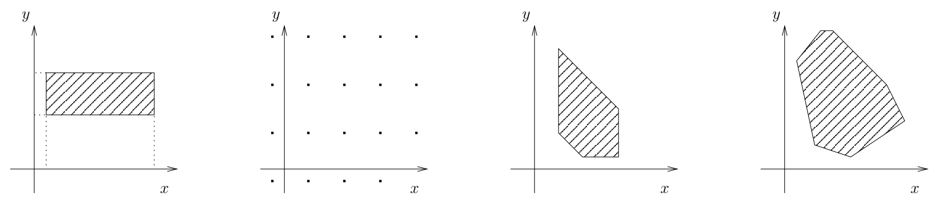
3.1.3 Numerical abstract domains

In Section 3.1.1, we left the domain for representing sets of stores as a parameter and only assumed such a domain to provide a series of "abstract operations". We discuss now common choices for the instantiation of this domain.

Numerical domains: A very large range of domains have been introduced for handling numerical constraints:

- **Non-relational domains** abstract each variable separately, such as:
  - intervals [CC77] (constraints of the form $a \leq x \leq b$, where $x \in \mathbb{X}$ and a, b are constants);
  - congruence [Gra89] (constraints of the form $x \in a\mathbb{Z} + b$, where $x \in \mathbb{X}$ and a, b are constants);
- **Relational domains** allow constraints involving several variables, at a higher cost, such as:
  - Karr domain [Kar76] expresses linear equalities among program variables, such as $a \star x + b \star y + c \star z + \ldots = c$;
  - polyhedra [CH78] handle linear inequalities among program variables, such as $a \star x + b \star y + c \star z + \ldots \leq c$;
  - octagons [Min04b] restrict to inequalities of the form $\pm x \pm y \leq c$ where $x, y \in \mathbb{X}$ and c is a constant;

Some examples are displayed in Figure 3.2



(a) Intervals (b) Congruences (c) Octagons (d) Polyhedra

Figure 3.2: A few numerical domains (for two variable environments)

Boolean abstractions: As in the case of numeric variables, we can use

- **Non-relational abstractions:** for instance, we can use $\mathcal{P}(\mathbb{B})$ so as to describe the set of possible values for a boolean variables;
- **Relational abstractions**, e.g. based on binary decision diagrams (BDDs) [Bry86].

Combining domains: First, the mapping of concrete variables into abstract memory locations should be addressed in general, when in presence of unbounded structures. The literature is this domain is rather broad: we can cite memory and domain combination [CL05], analyses targeted at inferring properties about the memory layout [SRW02].

Second, the abstract domain for representing sets of stores should usually account for various kinds of predicates evoked in the previous paragraphs:

- in some cases, a new domain can be obtained directly from a more simple one, as is the case of the relations among boolean and numerical values used in Astrée :

this domain inputs a domain for representing numerical values as a parameter (see above).
- in most cases, a product allows to build a new domain from several domains. Then, a reduction operation [CC79] is usually required so as to allow the constraints of one domain to refine the information in the other domains.

The following definition formalizes the notion of *reduced product*:

Definition 3.1.1. Reduced product.
Let $(D_0^\sharp, \gamma_0)$ and $(D_1^\sharp, \gamma_1)$ be two abstractions of a same concrete domain D.
Then, the product abstraction *$(D_p^\sharp, \gamma_p)$ is defined by:*
- $D_p^\sharp = D_0^\sharp \times D_1^\sharp$;
- $\forall (x_0, x_1) \in D_p^\sharp,\ \gamma_p(x_0, x_1) = \gamma_0(x_0) \cap \gamma_1(x_1)$.

A drawback of this domain is that distinct abstract element may have the same concretization; for instance, it is common that $\gamma_p(x_0, \bot) = \gamma_p(\bot, x_1) = \emptyset$
The reduced product *is the quotient of $D_p^\sharp$ for the relation $\mathcal{R}$ defined by:*

$$(x_0, x_1)\mathcal{R}(x'_0, x'_1) \iff \gamma_p(x_0, x_1) = \gamma_p(x'_0, x'_1)$$

In practice, only an approximation of it may be computed.

3.2 Denotational Abstraction

The *denotational* abstraction is one of the most common abstractions of sets of traces is; it basically amounts to forgetting all about the history of program execution, and keeping only some kind of relation between the original state and the final state of traces.

3.2.1 Denotational semantics

Abstraction into functions: The classical definition of denotational semantics [Sco70] introduces functions mapping initial states into final states, as a way do define the meaning of programs. Intuitively, it forgets about all intermediate states and collects the relation between initial and final states. Denotational semantics is an abstraction of the operational semantics [Cou97a].

In this thesis, we factor the control states out of the states, when using the denotational semantics, by partitioning this functional representation into sets of traces from a control state $l_\vdash$ to a control state $l_\dashv$ or into sets of traces following some paths in the control flow graph. This amounts to defining functions mapping memory states into sets of memory states (there may be several output states due to non-determinism). Therefore, we write $\mathfrak{Den}$ for the set of functions $\mathbb{M} \to \mathcal{P}(\mathbb{M})$ (we also make a slight abuse of notation and let $\circ$ be defined over $\mathfrak{Den}$ by $\forall \phi_0, \phi_1 \in \mathfrak{Den},\ \phi_1 \circ \phi_0 : \rho \mapsto \cup\{\phi_1(\rho') \mid \rho' \in \phi_0(\rho)\}$).

Hence, we define an abstraction of a set of traces into a function mapping a store into a set of stores, that throws away the control states, as follows:

Definition 3.2.1. Abstraction to function.
We let the functional abstraction of sets of traces be defined by:

$$
\begin{array}{rrcl}
 & (\mathcal{P}(\Sigma), \subseteq) & \xleftarrow[\alpha_{\mathcal{F}}]{\gamma_{\mathcal{F}}} & (\mathbb{M} \rightarrow \mathcal{P}(\mathbb{M}), \subseteq) \\
\alpha_{\mathcal{F}}: & \mathcal{P}(\Sigma) & \rightarrow & (\mathbb{M} \rightarrow \mathcal{P}(\mathbb{M})) \\
 & \mathcal{E} & \mapsto & \lambda(\rho_0 \in \mathbb{S}).\{\rho_n \mid \langle(\ell_0, \rho_0), \ldots, (\ell_n, \rho_n)\rangle \in \mathcal{E}\} \\
\gamma_{\mathcal{F}}: & (\mathbb{M} \rightarrow \mathcal{P}(\mathbb{M})) & \rightarrow & \mathcal{P}(\Sigma) \\
 & \Phi & \mapsto & \{\langle(\ell_0, \rho_0), \ldots, (\ell_n, \rho_n) \in \mathcal{E}\rangle \mid \rho_n \in \Phi(\rho_0)\}
\end{array}
$$

(we use the same notation for the pointwise ordering over $\mathbb{M} \rightarrow \mathcal{P}(\mathbb{M})$ *as for the conventional ordering over* $\mathcal{P}(\mathbb{M})$*).*

Note that this definition abstracts the initial and final control states away; more careful abstractions are presented in the following two subsections, by composing this abstraction with several abstractions that aim at defining what set of traces the abstraction of Definition 3.2.1 should be applied to.

Remark 3.2.1. Relational semantics and predicate transformers.
It has been observed in [Cou97a] that the denotational semantics is also equivalent to other common forms of semantics, including relational semantics [MT91], predicate transformer semantics [Dji75] We follow the denotational presentation for the sake of convenience.

Collecting sub-traces: Before we set up definitions of denotational semantics along paths or between control states in programs, we need to solve the following problems: our current definition of $[\![P]\!]$ collects traces starting from the initial points only; however, we need to collect all "sub-traces" in order to capture the behavior of P, say, between ℓ_0 and ℓ_1; otherwise, if ℓ_0 is not the entry point, we would not be able to isolate the "parts" of executions of P starting from ℓ_0.

Two traces σ_0, σ_1 such that the last state of σ_0 and the last state of σ_1 can be combined together in a single, longer execution trace:

Definition 3.2.2. Concatenation of traces, sub-trace.
Let $\sigma = \langle s_0, \ldots, s_n\rangle$ *and* $\sigma' = \langle s'_0, \ldots, s'_m\rangle$ *be two traces. If* $s_n = s'_0$*, we define the* concatenation $\sigma \frown \sigma'$ *of* σ *and* σ' *by:*

$$\sigma \frown \sigma' = \langle s_0, \ldots, s_n, s'_1, \ldots, s'_m\rangle$$

We let this operation be defined for sets of traces as well (and abusively use the same notation for the concatenation of sets of traces).

We say that σ is a sub-trace of σ' (and we write $\sigma \preccurlyeq \sigma'$) if and only if there exist two traces σ_0, σ_1 such that $\sigma' = \sigma_0 \frown \sigma \frown \sigma_1$.

We can now define what kind of sets of traces we are interested in:

Definition 3.2.3. closed set of traces.

Let $\mathcal{E}$ be a set of traces. We say that $\mathcal{E}$ is:

- closed *if and only if:*

$$\forall \sigma, \sigma', \in \Sigma \textit{ such that } (\sigma \frown \sigma') \textit{ is defined, } (\sigma \frown \sigma') \in \mathcal{E} \Longrightarrow (\sigma \in \mathcal{E} \wedge \sigma' \in \mathcal{E})$$

- strongly closed *if and only if:*

$$\forall \sigma, \sigma', \in \Sigma \textit{ such that } (\sigma \frown \sigma') \textit{ is defined, } (\sigma \frown \sigma') \in \mathcal{E} \Longleftrightarrow (\sigma \in \mathcal{E} \wedge \sigma' \in \mathcal{E})$$

We write $\mathfrak{C}[\Sigma]$ for the set of strongly closed sets of traces.

Intuitively a set of traces $\mathcal{E}$ is closed in the sense of Definition 3.2.3 if and only if it is closed under the $\preccurlyeq$ relation: if $\sigma \preccurlyeq \sigma'$ and $\sigma' \in \mathcal{E}$, then $\sigma \in \mathcal{E}$. Strongly closed sets of traces are also closed under concatenation.

We remark that it is possible to complete any set of traces into a closed set of traces:

Definition 3.2.4. Trace closure operator.

We let $\mathit{clos} : \mathcal{P}(\Sigma) \rightarrow \mathcal{P}(\Sigma)$ be the closure operator *defined by $\mathit{clos}(\mathcal{E}) = \{\sigma \in \Sigma \mid \exists \sigma' \in \Sigma,\ \sigma \preccurlyeq \sigma'\}$.*

Clearly, clos is an upper closure operator (it is extensive, monotone and idempotent), and $\forall \mathcal{E} \subseteq \Sigma$, $\mathit{clos}(\mathcal{E})$ is closed.

Finally, we can express the "*new*" semantics, which we are interested in by: $[\![P]\!]_{\mathbf{c}} = \mathit{clos}([\![P]\!])$; clearly, $[\![P]\!]_{\mathbf{c}}$ is closed. We note that $[\![P]\!]_{\mathbf{c}}$ is strongly closed: if σ and σ' are two traces of P, which can be concatenated, then $\sigma \frown \sigma' \in [\![P]\!]_{\mathbf{c}}$.

This new semantics can be written as a least fixpoint as well. In fact, this new semantics is equivalent to $[\![P]\!]$: we can write a Galois-bijection which relate Σ and $\mathfrak{C}[\Sigma]$, and prove the equivalence between $[\![P]\!]$ and $[\![P]\!]_{\mathbf{c}}$ by a trivial fixpoint transfer arguments (Theorem 2.3.2).

In the following, we may simply write $[\![P]\!]$ for $[\![P]\!]_{\mathbf{c}}$ (and mention that we are using the strongly closed semantics), since both semantics express the same behaviors. Of course, we consider the closed version in this section.

3.2.2 Functions "From-To"

We introduce an abstraction that keeps only the traces between two control points:

Definition 3.2.5. From-To abstraction.

For any pair of control points $l_{\vdash}, l_{\dashv} \in \mathbb{L}$, we define the following Galois-connection:

$$\begin{tikzcd}(\mathcal{P}(\Sigma), \subseteq) \arrow[r, shift left, "\alpha_{t[l_{\vdash}, l_{\dashv}]}"'] & (\mathcal{P}(\Sigma), \subseteq) \arrow[l, shift left, "\gamma_{t[l_{\vdash}, l_{\dashv}]}"']\end{tikzcd}$$

$$\begin{array}{rrcl} \alpha_{t[l_{\vdash}, l_{\dashv}]} : & \mathcal{P}(\Sigma) & \rightarrow & \mathcal{P}(\Sigma) \\ & \mathcal{E} & \mapsto & \{\langle (l_0, \rho_0), \ldots, (l_n, \rho_n) \rangle \in \mathcal{E} \mid l_0 = l_{\vdash} \wedge l_n = l_{\dashv}\} \end{array}$$

This defines a family of Galois-connections (i.e., one Galois-connection for each pair $(l_{\vdash}, l_{\dashv})$).

A first partitioned denotational semantics is obtained by composing this abstraction with the functional abstraction introduced in the previous subsection:

Definition 3.2.6. Functional, From-To abstraction.

For any pair of control points $l_{\vdash}, l_{\dashv} \in \mathbb{L}$, we define the following Galois-connection:

$$\begin{tikzcd}(\mathcal{P}(\Sigma), \subseteq) \arrow[r, shift left, "\alpha_{t\mathcal{F}[l_{\vdash}, l_{\dashv}]}"'] & (\mathfrak{Den}, \subseteq) \arrow[l, shift left, "\gamma_{t\mathcal{F}[l_{\vdash}, l_{\dashv}]}"']\end{tikzcd}$$

$$\begin{array}{rcl} \alpha_{t\mathcal{F}[l_{\vdash}, l_{\dashv}]} & = & \alpha_{\mathcal{F}} \circ \alpha_{t[l_{\vdash}, l_{\dashv}]} \\ \gamma_{t\mathcal{F}[l_{\vdash}, l_{\dashv}]} & = & \gamma_{t[l_{\vdash}, l_{\dashv}]} \circ \gamma_{\mathcal{F}} \end{array}$$

This abstraction is mostly useful when we need to consider only the effect of a piece of code on the memory state. In particular, we will use this kind of abstractions in order to define dependences induced by fragments of programs and to reason about the semantic equivalence of programs.

Definition 3.2.1. From-To semantics.

Let us consider the P program below:

$$\begin{array}{ll} l_0 : & \mathbf{if}(x < 4)\,\{ \\ l_1 : & \quad x = 4; \\ l_2 : & \}\,\mathbf{else}\,\{ \\ l_3 : & \quad y = x + 3; \\ l_4 : & \} \\ l_5 : & \ldots \end{array}$$

Then, the semantics $\alpha_{t\mathcal{F}[l_0, l_5]}[\![P]\!]$ maps initial stores into final stores; for instance if $\rho(x) = 0$, then $\alpha_{t\mathcal{F}[l_0, l_5]}[\![P]\!](\rho) = \{\rho[x \leftarrow 4]\}$.

3.2.3 Functions "Along paths"

Path in the control flow: A *path* p is a sequence of control states $l_0 \cdot l_1 \cdot \ldots \cdot l_n$. The length of such a path is n (number of control states involved minus one or equivalently, number of edges considered) and is denoted $\mathbf{len}(p)$. We write $\mathcal{P}(l_\vdash, l_\dashv)$ for the set of paths from $l_\vdash$ to $l_\dashv$.

The semantics of s restricted to a path is the set of traces in $[\![s]\!]$ that follow this path; it defines an abstraction of the standard, trace semantics:

Definition 3.2.7. Path abstraction.

For all path p, we let the path abstraction be defined by the following Galois-connection:

$$\mathcal{P}(\Sigma) \underset{\alpha_{p[p]}}{\overset{\gamma_{p[p]}}{\leftrightarrows}} \mathcal{P}(\Sigma)$$

$$\begin{array}{rccl} \alpha_{p[p]}: & \mathcal{P}(\Sigma) & \rightarrow & \mathcal{P}(\Sigma) \\ & \mathcal{E} & \mapsto & \{\langle (l_0, \rho_0), \ldots, (l_n, \rho_n)\rangle \in \mathcal{E} \mid p = l_0 \cdot l_1 \cdot \ldots l_n\} \end{array}$$

Similarly as in the last subsection, we can apply the abstraction of traces into functions to this semantics:

Definition 3.2.8. Functional, path abstraction.

For any path p, we define the following Galois-connection:

$$(\mathcal{P}(\Sigma), \subseteq) \underset{\alpha_{p\mathcal{F}[p]}}{\overset{\gamma_{p\mathcal{F}[p]}}{\leftrightarrows}} (\mathfrak{Den}, \subseteq)$$

$$\begin{array}{rcl} \alpha_{p\mathcal{F}[p]} & = & \alpha_{\mathcal{F}} \circ \alpha_{p[p]} \\ \gamma_{p\mathcal{F}[p]} & = & \gamma_{p[p]} \circ \gamma_{\mathcal{F}} \end{array}$$

It defines a Galois-connection for each path.

This abstraction is useful when we need to isolate the behavior of programs on *some* path(s) e.g., in order to prove some semantic equivalence between paths in different programs.

The restriction of the semantics to paths allows to partition the from-to semantics, as shown in the following lemma:

Lemma 3.2.1. Partitioning of the graph-denotational semantics.

Let $l_\vdash, l_\dashv \in \mathbb{L}$. Then, for any set of traces $\mathcal{E} \subseteq \Sigma$,

$$\alpha_{t[l_\vdash, l_\dashv]}(\mathcal{E}) = \bigcup\{\alpha_{p[p]}(\mathcal{E}) \mid p \in \mathcal{P}(l_\vdash, l_\dashv)\}$$

Moreover, if p, p' are two distinct paths, then, clearly $\alpha_{p[p]}(\mathcal{E}) \cap \alpha_{p[p']}(\mathcal{E}) = \emptyset$. Furthermore, for any $\rho \in \mathbb{M}$,

$$\alpha_{t\mathcal{F}[l_\vdash, l_\dashv]}(\mathcal{E})(\rho) = \bigcup\{\alpha_{p\mathcal{F}[p]}(\mathcal{E})(\rho) \mid p \in \mathcal{P}(l_\vdash, l_\dashv)\}$$

(in other words $\{\alpha_{p\mathcal{F}[p]}(\mathcal{E})(\rho) \mid p \in \mathcal{P}(\ell_{\vdash}, \ell_{\dashv})\}$ *partitions* $\alpha_{t\mathcal{F}[\ell_{\vdash},\ell_{\dashv}]}(\mathcal{E})(\rho)$ *since the elements of this set are pairwise disjoint.)*

Proof.

Straightforward. □

Definition 3.2.2. Path semantics.

We consider the program of Example 3.2.1, and the semantics between ℓ_0 *and* ℓ_5*. There are two paths between these two points:* $p_{\mathrm{t}} = \ell_0 \cdot \ell_1 \cdot \ell_2 \cdot \ell_5$ *and* $p_{\mathrm{f}} = \ell_0 \cdot \ell_3 \cdot \ell_4 \cdot \ell_5$*. Let* $\rho \in \mathbb{M}$*, such that* $\rho(x) = 0$*. Then:*

- $\alpha_{p\mathcal{F}[p_{\mathrm{t}}]}(\llbracket P \rrbracket)(\rho) = \{\rho[x \leftarrow 4]\}$*;*
- $\alpha_{p\mathcal{F}[p_{\mathrm{f}}]}(\llbracket P \rrbracket)(\rho) = \emptyset$*.*

3.2.4 Composition

We now relate two natural operations:

- the concatenation of traces;
- the composition of functions.

Composition as an approximation of composition: We propose a characterization of the composition of the semantics along paths for closed sets of traces:

Lemma 3.2.2. Composition along paths.

Let $\mathcal{E}$ *be a* closed *set of traces. Let* $p = \ell_0 \cdot \ldots \cdot \ell_n \cdot \ldots \cdot \ell_m$ *be a path. We let* $p' = \ell_0 \cdot \ldots \cdot \ell_n$ *and* $p'' = \ell_n \cdot \ldots \cdot \ell_m$*. Then:*

$$\alpha_{p\mathcal{F}[p]}(\mathcal{E}) \subseteq \alpha_{p\mathcal{F}[p'']}(\mathcal{E}) \circ \alpha_{p\mathcal{F}[p']}(\mathcal{E})$$

In case $\mathcal{E}$ *is* strongly closed*:*

$$\alpha_{p\mathcal{F}[p]}(\mathcal{E}) = \alpha_{p\mathcal{F}[p'']}(\mathcal{E}) \circ \alpha_{p\mathcal{F}[p']}(\mathcal{E})$$

Proof.

(case where $\mathcal{E}$ is closed). Let $\rho, \rho'' \in \mathbb{M}$. Then:

$$\begin{array}{rl}
\multicolumn{2}{l}{\rho'' \in \alpha_{p\mathcal{F}[p]}(\mathcal{E})(\rho)} \\
\iff & \exists \langle (\ell_0, \rho_0), \ldots, (\ell_n, \rho_n), \ldots, (\ell_m, \rho_m) \rangle \in \mathcal{E},\ \rho_0 = \rho \wedge \rho_m = \rho'' \\
\iff & \exists \sigma' = \langle (\ell_0, \rho_0), \ldots, (\ell_n, \rho_n) \rangle, \sigma'' = \langle (\ell_n, \rho_n), \ldots, (\ell_m, \rho_m) \rangle, \\
& \quad \sigma' \frown \sigma'' \in \mathcal{E} \wedge \rho_0 = \rho \wedge \rho_m = \rho'' \\
\Longrightarrow & \exists \langle (\ell_0, \rho_0), \ldots, (\ell_n, \rho_n) \rangle, \langle (\ell_n, \rho_n), \ldots, (\ell_m, \rho_m) \rangle \in \mathcal{E},\ \rho_0 = \rho \wedge \rho_m = \rho'' \\
& \quad \text{since } \mathcal{E} \text{ is closed} \\
\iff & \exists \rho' \in \mathbb{M},\ \exists \langle (\ell_0, \rho_0), \ldots, (\ell_n, \rho_n) \rangle, \langle (\ell_n, \rho_n), \ldots, (\ell_m, \rho_m) \rangle \in \mathcal{E}, \\
& \quad \rho_0 = \rho \wedge \rho_n = \rho' \wedge \rho_m = \rho'' \\
\iff & \exists \rho' \in \alpha_{p\mathcal{F}[p']}(\mathcal{E})(\rho),\ \rho'' \in \alpha_{p\mathcal{F}[p'']}(\mathcal{E})(\rho') \\
\iff & \rho'' \in \alpha_{p\mathcal{F}[p'']}(\mathcal{E}) \circ \alpha_{p\mathcal{F}[p']}(\mathcal{E})(\rho)
\end{array}$$

which concludes the proof.
In case $\mathcal{E}$ is strongly closed, the implication in the middle of the proof can be turned into an equivalence. □

Closed sets of traces and fixpoint-definitions: The following theorem shows that a strongly closed set of traces can be described by a least fixpoint equation:

Theorem 3.2.3. Strongly set of traces as a least fixpoint.
Let $\mathcal{E} \subseteq \Sigma$. Then, there exists $F : \Sigma \rightarrow \Sigma$ and a set $\mathcal{I} \subseteq \Sigma$ such that:
- *if $\mathcal{E}$ is closed, then $\mathcal{E} \subseteq \mathbf{lfp}_{\mathcal{I}} F$*
- *if $\mathcal{E}$ is strongly closed, then $\mathcal{E} = \mathbf{lfp}_{\mathcal{I}} F$.*

Proof.

We let:
- $\mathcal{I} = \mathcal{E} \cap \{\langle s \rangle \mid s \in \mathbb{S}\}$;
- $\mathcal{R}$ is the relation $\{(s, s') \in \mathbb{S}^2 \mid \langle s, s' \rangle \in \mathcal{E}\}$.

We let:

$$\begin{array}{rlcl}
F : & \mathcal{P}(\Sigma) & \rightarrow & \mathcal{P}(\Sigma) \\
& E & \mapsto & \mathcal{I} \cup \{\langle s_0, \ldots, s_n, s_{n+1} \rangle \mid \langle s_0, \ldots, s_n \rangle \in E \wedge (s_n, s_{n+1}) \in \mathcal{R}\}
\end{array}$$

Clearly, F is continuous, so $\mathbf{lfp} F = \bigcup_{n \in \mathbb{N}} F^n(\emptyset)$. Let us assume that $\mathcal{E}$ is closed. Then, we can show by induction on the length of σ that $\sigma \in \mathcal{E} \Longrightarrow \sigma \in F^n(\emptyset)$:
- if $n = 1$, then $\sigma \in \mathcal{I} = F^0(\emptyset)$;
- if $n \geq 1$ and the property is holds for n, and σ has length $n + 1$, then $\sigma = \sigma' \frown \sigma''$, where σ' has length n and $\sigma'' = \langle s_{n-1}, s_n \rangle$ has length 2. Therefore, the induction hypothesis implies that $\sigma' \in F^n(\emptyset)$; moreover, $(s_{n-1}, s_n) \in \mathcal{R}$; as a consequence, $\sigma \in F^{n+1}(\emptyset)$.

As a consequence, $\sigma \in \mathbf{lfp}F$.
The proof of the converse implication is similar. $\square$

This theorem could be used as a basis in order to relate fixpoint definitions for trace semantics and denotational semantics. We shall use it in order to provide fixpoint definitions for semantics derived by applying some abstractions to more simple semantics.

3.2.5 Static analysis

We propose here to build a denotational abstract interpreter from the denotational semantics, that gives similar results as the simple interpreter described in Section 3.1.2. We use the same notations.

Definition and soundness of the interpreter: Let $s \in \mathbb{s}$ be a statement; we write $\ell_\vdash$ (resp. $\ell_\dashv$) for the control point before (resp. after) s. We let the denotational semantics of s be the function $[\![s]\!]_\delta = \alpha_{t\mathcal{F}[\ell_\vdash,\ell_\dashv]}([\![s]\!])$. The abstract semantics of s is the function $[\![s]\!]^\sharp : D^\sharp_{\mathbb{M}} \rightarrow D^\sharp_{\mathbb{M}}$, which inputs an abstract pre-condition and returns a strongest post-condition. It should be sound in the sense that the output of the abstract semantics should over-approximate the set of output states of the underlying, concrete denotational semantics.

We propose on Figure 3.3 the definition of a very simple denotational semantics-based interpreter.

statement s	abstract semantics
$x := e;$	$[\![s]\!]^\sharp : d \mapsto \mathit{assign}(x, e, d)$
$\mathbf{if}(e)\{s_0\}\mathbf{else}\{s_1\}$	$[\![s]\!]^\sharp : d \mapsto [\![s_0]\!]^\sharp(\mathit{guard}(e, \mathbf{true}, d)) \sqcup [\![s_1]\!]^\sharp(\mathit{guard}(e, \mathbf{false}, d))$
$\mathbf{while}(e)\{s\}$	$[\![s]\!]^\sharp : d \mapsto \mathit{guard}(e, \mathbf{false}, \mathbf{lfp}^\sharp F^\sharp)$ where $F^\sharp : \begin{array}{ccc} D^\sharp_{\mathbb{M}} & \rightarrow & D^\sharp_{\mathbb{M}} \\ d_0 & \mapsto & d_0 \sqcup [\![s]\!]^\sharp(\mathit{guard}(e, \mathbf{true}, d)) \end{array}$ and $\mathbf{lfp}^\sharp$ computes an abstract post-fixpoint
$\mathbf{input}(x \in V);$	$[\![s]\!]^\sharp : d \mapsto \mathit{guard}(x \in V^\sharp, \mathit{forget}(x, d))$
$\mathbf{assert}(e);$	$[\![s]\!]^\sharp : d \mapsto \mathit{guard}(e, \mathbf{true}, d)$

Figure 3.3: A simple abstract interpreter

The abstract semantics displayed in Figure 3.3 is sound:

Theorem 3.2.4. Soundness of the analysis.

The abstract semantics soundly approximates the denotational semantics:

$$\forall \rho, \rho' \in \mathbb{M},\ d \in D^\sharp_{\mathbb{M}},\ \rho \in \gamma_{\mathbb{M}}(d) \wedge \rho' \in [\![s]\!]_\delta(\rho) \Longrightarrow \rho' \in \gamma_{\mathbb{M}}([\![s]\!]^\sharp(d))$$

Proof.

By induction on the structure of the code.
The case of the loop is based on the soundness of the $\mathbf{lfp}^\sharp$ operator; in practice, it is derived from a widening operator $\nabla_{\mathbb{M}}$ over $D^\sharp_{\mathbb{M}}$, so the soundness and termination of $\mathbf{lfp}^\sharp$ follow from Section 2.3.3. □

As a corrolary, the abstract semantics is sound with respect to the standard, operational semantics. Indeed, if $\ell_\vdash : s; \ell_\dashv$ is a program, then:

$$\forall \langle (\ell_\vdash, \rho_\vdash), \ldots, (\ell_\dashv, \rho_\dashv) \rangle \in [\![s]\!],\ \forall d \in D^\sharp_{\mathbb{M}},\ \rho_\vdash \in \gamma_{\mathbb{M}}(d) \Longrightarrow \rho_\dashv \in \gamma_{\mathbb{M}}([\![s]\!]^\sharp(d))$$

Comparison with iterations over a control flow graph: This approach is currently used in ASTRÉE and presents many advantages, due to the fact that no global iteration strategy should be implemented (since the abstract interpretation proceeds recursively on the syntax of the programs):

- This abstract semantics is based on an **efficient iteration strategy**. In particular, no work-list or other strategy algorithm is needed, since the strategy is fully defined by the control flow of the programs. Moreover, the order abstract transfer functions are applied in is optimal in the sense that the issue mentioned in Example 3.1.1 never occurs.
- This approach requires no local invariant storage, except for the computation of loop invariants with $\mathbf{lfp}^\sharp$; in practice, the analyzer need to keep one invariant at the head of each loop while analyzing its body.
- The set of widening point is also completely defined; it corresponds to loop heads.

Computation of an invariant over $D^\sharp$: We propose to derive from the interpreter in Figure 3.3 an abstract interpreter computing an invariant in $D^\sharp = \mathbb{L} \to D^\sharp_{\mathbb{M}}$: we still wish to get a local invariant for each control state as a result of the analysis.

Similarly, we may be interested in other outputs from the analyzer, such as alarm reports, in case the result of the analysis does not prove all critical operations safe.

We propose a "two-modes" analyzer:

- a *Check* mode, for any phase in the analysis *except* iterations in loops before a post-fixpoint is reached (i.e., before an over-approximation of the concrete states is reached); all analysis side effects should be performed in this mode;
- an *Iter* mode, for the post-fixpoint iterations in loops; this mode does not carry out any analysis side effect (computation of the final, safe local abstract invariants or alarm reports).

This way, we can note that the analyzer interprets any statement exactly once in $\mathcal{Check}$ mode (the case of interprocedural programs requires taking into account all the calls to the function).

The delay of all side effects to the last phase of the analysis is important for several reasons. First, some earlier iterations may involve *less precise* invariants if the analyzer computes a sequence of decreasing iterations (Remark 2.3.1) in the end, so that the alarms or exported invariants would be less precise (more false alarms, or worse invariants than the one actually available in the end of the analysis). Second, the export of local invariants requires a lot of memory, so it is practically preferable to delay it to the end of the analysis.

We let $\mathcal{M}$ denote $\{\mathcal{Check}, \mathcal{Iter}\}$. We write $[\![P]\!]^\sharp_{\mathcal{M}}$ for the *extended* abstract interpreter; it is a function from $\mathcal{M} \times D^\sharp \times D^\sharp_{\mathbb{M}}$ into $\mathcal{M} \times D^\sharp \times D^\sharp_{\mathbb{M}}$ (we focus on the computation of the approximation of all reachable states; the computation of a superset of alarms would be similar). The definition of this extended abstract interpreter follows the rules in Figure 3.3. Here are two rules:

- case of an *assignment* $l_\vdash : x := e; l_\dashv$:

$$\begin{array}{rlll}[\![s]\!]^\sharp_{\mathcal{M}} : & (\mathcal{Iter}, d, \mathfrak{I}) & \mapsto & (\mathcal{Iter}, \mathcal{assign}(x, e, d), \mathfrak{I}) \\ & (\mathcal{Check}, d, \mathfrak{I}) & \mapsto & (\mathcal{Check}, d', \mathfrak{I}')\text{where} \left\{\begin{array}{l} d' = \mathcal{assign}(x, e, d) \\ \mathfrak{I}'(l_\dashv) = d' \\ \mathfrak{I}'(l) = \mathfrak{I}(l) \text{ if } l \neq l_\dashv \end{array}\right.\end{array}$$

- case of a *loop* $l_\vdash$: **while**$(e)\{s\}; l_\dashv$:
  - in $\mathcal{Iter}$ mode (loop in another loop), then: $[\![s]\!]^\sharp(\mathcal{Iter}, d, \mathfrak{I}) = (\mathcal{Iter}, \mathcal{guard}(e, \mathbf{false}, \mathbf{lfp}^\sharp F^\sharp), \mathfrak{I})$
  - in $\mathcal{Check}$ mode, then we let $d \in D^\sharp_{\mathbb{M}}$, $\mathfrak{I} \in D^\sharp$ and let $d', \mathfrak{I}'$ be defined by

$$\begin{array}{rrcl} & (\mathcal{Iter}, d', \mathfrak{I}') = & \mathbf{lfp}^\sharp F^\sharp & \\ F^\sharp : & D^\sharp_{\mathbb{M}} & \rightarrow & D^\sharp_{\mathbb{M}} \\ & d_0 & \mapsto & d_0 \sqcup [\![s]\!]^\sharp(\mathcal{guard}(e, \mathbf{true}, d)) \end{array}$$

    We also write $d'' = \mathcal{guard}(e, \mathbf{false}, d')$, and $\mathfrak{I}''$ derived from $\mathfrak{I}'$ by $\mathfrak{I}''(l_\dashv) = d''$ Then: $[\![s]\!]^\sharp(\mathcal{Check}, d, \mathfrak{I}) = (\mathcal{Check}, d'', \mathfrak{I}'')$.

We could extend Theorem 3.2.4, by proving that this interpreter not only computes a sound output invariant, but also a sound over-approximation of all reachable states for the given input invariant (just as in Section 3.1) (or a safe superset of alarms).

3.2.6 Symbolic Representation

The denotational semantics, which we introduced in Section 3.2.1 is not computable: it does not provide a more convenient way to represent the functions mapping initial stores to final stores for a piece of program than the program itself. We propose here a way of doing so, which is based on symbolic transfer functions [CL96].

Expressions:

$$e(\in \mathbb{e}_{\mathbb{S}}) \;::=\; \ldots \mid \mathbf{is\_alias}(l, l')\ (\text{where } l, l' \in \mathbb{l}) \mid \mathbf{rnd}(V)\ (\text{where } V \subseteq \mathbb{V})$$

Symbolic transfer functions:

$$\begin{array}{rcll} \delta(\in \mathbb{S}) & ::= & \Box & \\ & \mid & \lfloor x_0 \leftarrow e_0, \ldots, x_n \leftarrow e_n \rfloor & x_0, \ldots, x_n \in \mathbb{X}, e_0, \ldots, e_n \in \mathbb{e} \\ & & & \forall i, j,\ i \neq j \Rightarrow x_i \neq x_j \\ & \mid & \lfloor e \ ?\ \delta_0 \mid \delta_1 \rfloor & e \in \mathbb{e}, \delta_0, \delta_1 \in \mathbb{S} \end{array}$$

Figure 3.4: Grammar of symbolic transfer functions

Syntax: A *symbolic transfer* function is:

- either the "void" function $\Box$, which denotes the absence of transition (blocking function);
- or a parallel assignment $\lfloor x_0 \leftarrow e_0, \ldots, x_n \leftarrow e_n \rfloor$ where $\forall i, j,\ i \neq j \Longrightarrow x_i \neq x_j$;
- or a conditional $\lfloor e \ ?\ \delta_0 \mid \delta_1 \rfloor$ where e is an expression and δ_0, δ_1 are symbolic transfer functions.

We write $\mathbb{S}$ for the set of symbolic transfer functions. We note that the empty assignment does not modify the content of the memory, and just returns the input store; hence, it corresponds to the identity function; we will write ι for it.

The requirement that the l-values in the parallel assignment should be pairwise distinct is crucial for the semantics of the function to be properly defined. In practice, we always make sure to define only symbolic functions that fulfill this requirement.

In the following, we assume that the expressions in symbolic transfer functions provide two additional features:

- alias testing: $\mathbf{is\_alias}(l, l')$ (where $l, l' \in \mathbb{l}$) returns **true** if l and l' evaluates to the same memory location and returns **false** otherwise (this operator allows to guarantee that all l-values in a parallel assignment should be distinct, by introducing alias testing);
- non-determinism: $\mathbf{rnd}(()V)$ returns any value in V (where $V \subseteq \mathbb{V}$).

The full grammar of symbolic transfer functions is displayed in Figure 3.4.

Semantics: The semantics of expressions is defined straightforwardly. Note that the semantics of an expression $e \in \mathbb{e}_{\mathbb{S}}$ is a function $[\![e]\!] : \mathbb{M} \to \mathcal{P}(\mathbb{V})$, since we allowed non-determinism.

Intuitively, a symbolic transfer function δ denotes a store transformer; hence, the semantics of a symbolic transfer function $\delta \in \mathbb{S}$ is a function $[\![\delta]\!] : \mathbb{M} \to \mathcal{P}(\mathbb{M})$. We let it be defined as follows:

- $\forall \rho \in \mathbb{M},\ [\![\Box]\!](\rho) = \emptyset$;
- Let $\rho \in \mathbb{M}$, $l_0, \ldots, l_n \in \mathbb{l}_{\mathbb{S}}, e_0, \ldots, e_n \in \mathbb{e}_{\mathbb{S}}$, and $\forall i,\ x_i = [\![l_i]\!](\rho)$ and $V_i = [\![e_i]\!](\rho)$. Then, if $\forall i, j,\ i \neq j \Longrightarrow x_i \neq x_j$, then

$$[\![\lfloor l_0 \leftarrow e_0, \ldots, l_n \leftarrow e_n \rfloor]\!](\rho) = \{\rho[x_0 \leftarrow v_0, \ldots, x_n \leftarrow v_n] \mid \forall i,\ v_i \in V_i\}$$

- If $e \in \mathbb{e}_{\mathbb{S}}$, $\delta_0, \delta_1 \in \mathbb{S}$, then

$$[\![\lfloor e \mathbin{?} \delta_0 \mid \delta_1 \rfloor]\!](\rho) = \begin{cases} [\![\delta_0]\!](\rho) & \text{if } [\![e]\!](\rho) = \{\mathbf{true}\} \\ [\![\delta_1]\!](\rho) & \text{if } [\![e]\!](\rho) = \{\mathbf{false}\} \\ [\![\delta_0]\!](\rho) \cup [\![\delta_1]\!](\rho) & \text{if } [\![e]\!](\rho) = \{\mathbf{true}, \mathbf{false}\} \end{cases}$$

(note that there is no case $[\![e]\!](\rho) = \emptyset$; intuitively, the evaluation of the semantics of any expression for any store should contain at least one value).

Symbolic transfer functions-based definition: We propose on Figure 3.5 the symbolic transfer functions corresponding to all one-step transitions, for the simple language we introduced in Section 2.2.3. This definition simply mimics the transition rules provided in Figure 2.1(b). The soundness of the encoding writes down as follows:

$$\forall \ell, \ell' \in \mathbb{L}, \forall \rho, \rho' \in \mathbb{M}, (\ell, \rho) \rightarrow (\ell', \rho') \iff \rho' \in [\![\delta_{\ell,\ell'}]\!](\rho)$$

Remark 3.2.2. Errors.

If a statement $\ell_0 : s; \ell_1 : \ldots$ causes an error, then the corresponding transition between ℓ_0 and ℓ_1 is described by the transfer function $\square$ (blocking situation).
We may also choose to define explicitly the transitions from ℓ_1 to Ω with a transfer function $\delta_{\ell_0,\Omega}$; we choose not to define these transitions explicitly.

Composition and semantics along paths or sets of paths: A *syntactic composition operator* $\oplus : \mathbb{S} \times \mathbb{S} \rightarrow \mathbb{S}$ is defined for this language, such that:

$$\forall \delta_0, \delta_1 \in \mathbb{S}, [\![\delta_1 \oplus \delta_0]\!] = [\![\delta_1]\!] \circ [\![\delta_0]\!]$$

Basically, this operator:

- substitutes in the expressions that appear in δ_1 the l-values assigned in δ_0 with the assigned values;
- stacks the conditions from δ_0 and δ_1 (this corresponds to a kind of product);
- handles possible aliasing problems by inserting tests of the form **is_alias**(l, l') (where l and l' are l-values which maybe aliased) in order to carry out sound memory updates.

The soundness of such an operator is described in details and proved in, e.g. [Col96].

We can note that $\forall \delta \in \mathbb{S}$, $\iota \circ \delta = \delta \circ \iota = \delta$, so ι indeed is an identity element for $\circ$.

Another important point is that symbolic simplifications may take place either when computing the composition of a series of symbolic transfer functions or at any time (before, after, or in the middle of the composition of functions), by applying any computable simplification function $\mathit{simplify} : \mathbb{S} \rightarrow \mathbb{S}$, such that $\forall \delta \in \mathbb{S}$, $[\![\mathit{simplify}(\delta)]\!] = [\![\delta]\!]$ (and $\mathit{simplify}(\delta)$ is simpler to analyze, to compose, or for other tasks). Among the simplifications one may envisage, we can cite:

assignment	$\ell_0 : x := e; \ell_1$
	$\delta_{\ell_0,\ell_1} = \lfloor x \leftarrow e \rfloor$
conditional	$\ell_0 : \mathbf{if}(e)\, \{\ell_0^t : s_t; \ell_1^t\}\, \mathbf{else}\, \{\ell_0^f : s_f; \ell_1^f\}\, \ell_1$
	$\delta_{\ell_0,\ell_0^t} = \lfloor e \,?\, \iota \mid \Box \rfloor$
	$\delta_{\ell_0,\ell_0^f} = \lfloor e \,?\, \Box \mid \iota \rfloor$
	$\delta_{\ell_1^t,\ell_1} = \iota$
	$\delta_{\ell_1^f,\ell_1} = \iota$
loop	$\ell_0 : \mathbf{while}(e)\, \{\ell_0^b : s_t; \ell_1^b\}\, \ell_1$
	$\delta_{\ell_0,\ell_0^b} = \lfloor e \,?\, \iota \mid \Box \rfloor$
	$\delta_{\ell_0,\ell_1} = \lfloor e \,?\, \Box \mid \iota \rfloor$
	$\delta_{\ell_1^b,\ell_0} = \iota$
input	$\ell_0 : \mathbf{input}(x \in V); \ell_1$
	$\delta_{\ell_0,\ell_1} = \lfloor x \leftarrow \mathbf{rnd}(V) \rfloor$
assertion	$\ell_0 : \mathbf{assert}(e); \ell_1$
	$\delta_{\ell_0 \ell_1} = \lfloor e \,?\, \iota \mid \Box \rfloor$

Figure 3.5: Semantics defined with symbolic transfer functions

- the boolean simplifications due to assignments followed by condition testings;
- the removal of redundant **is_alias** expressions (with might be simplified in **true** or **false** thanks to a trivial alias analysis);
- various arithmetic simplifications, depending on data-types: for instance, $x - x = 0$, $x + x = 2x$ and $x + (y + z) = (x + y) + z$ hold for modular integer arithmetic; however the latter identity does not hold in floating point computations (indeed, the "+" operator is not associative due to the overflows).

At this point, we can use symbolic transfer functions in order to define the denotational semantics along a path:

Lemma 3.2.5. Semantics on a path.

We consider a program s and let $\mathcal{E} = [\![s]\!]$. Let $p = \ell_0 \cdot \ell_1 \cdot \ldots \cdot \ell_n$ be a path. Then,

$$\alpha_{p\mathcal{F}[p]}(\mathcal{E}) = [\![\delta_{\ell_{n-1},\ell_n} \oplus \delta_{\ell_{n-2},\ell_{n-1}} \oplus \ldots \oplus \delta_{\ell_0,\ell_1}]\!]$$

Proof.

The proof is done by induction on the length of the path:

- case of a path of length 0: $\alpha_{p\mathcal{F}[\ell_0]}(\mathcal{E}) = [\![\iota]\!]$

- case of a path of length 1: $\alpha_{p\mathcal{F}[l_0 \cdot l_1]}(\mathcal{E}) = [\![\delta_{l_0,l_1}]\!]$, by definition of δ_{l_0,l_1};
- case of a path of length $n+1$ (we assume the property holds for paths of length lesser than n):
  We let $p = l_0 \cdot \ldots \cdot l_{n+1}$ be a path of length $n+1$; we write p' for $l_0 \cdot \ldots \cdot l_n$, and $p'' = l_n \cdot l_{n+1}$. The induction assumption states that $\alpha_{p\mathcal{F}[p']}(\mathcal{E}) = \delta_{l_{n-1},l_n} \oplus \ldots \oplus \delta_{l_0,l_1}$. Then:

$$\begin{array}{rcll} \alpha_{p\mathcal{F}[p]}(\mathcal{E}) & = & \alpha_{p\mathcal{F}[p'']}(\mathcal{E}) \circ \alpha_{p\mathcal{F}[p']}(\mathcal{E}) & \text{by Lemma 3.2.2} \\ & = & [\![\delta_{l_n,l_{n+1}}]\!] \circ [\![\delta_{l_{n-1},l_n} \oplus \ldots \oplus \delta_{l_0,l_1}]\!] & \text{by induction hypothesis} \\ & = & [\![\delta_{l_n,l_{n+1}} \oplus \delta_{l_{n-1},l_n} \oplus \ldots \oplus \delta_{l_0,l_1}]\!] & \text{syntactic composition} \end{array}$$

This concludes the proof. □

We may also be interested in the denotational semantics defined for a collection of paths, which would be defined as the join of the semantics over each paths. We propose a result for finite sets of paths starting from a single point (sets of paths which do not start from the same point are not relevant in practice):

Lemma 3.2.6. Semantics over finite sets of paths.

Let $l_0 \in \mathbb{L}$ and $\mathcal{P}$ be a finite set of paths starting from l_0, such that $p \in \mathcal{P}$ implies that no prefix of p belongs to $\mathcal{P}$ (i.e., $\mathcal{P}$ can be see a set of paths in a tree, from the root to the leaves; in particular, $\mathcal{P}$ does not contain a path to an inner-node *of the tree). We let $\alpha_{p\mathcal{F}[\mathcal{P}]}(\mathcal{E})$ be defined by:*

$$\begin{array}{rrcl} \alpha_{p\mathcal{F}[\mathcal{P}]}(\mathcal{E}) : & \mathbb{M} & \rightarrow & \mathcal{P}(\mathbb{M}) \\ & \rho & \mapsto & \bigcup\{\alpha_{p\mathcal{F}[p]}(\mathcal{E})(\rho) \mid p \in \mathcal{P}\} \end{array}$$

Then, there exists a symbolic transfer function δ such that $\alpha_{p\mathcal{F}[\mathcal{P}]}(\mathcal{E}) = [\![\delta]\!]$.

Proof.

The proof relies on the assumption made on the structure of $\mathcal{P}$: it can be seen as a tree with root l_0; a path $p \in \mathcal{P}$ represents a branch inside the tree, starting at the roof, ending at a leaf. Hence, the proof can be done by induction on the depth of the tree underlying $\mathcal{P}$ and by case analysis over the statement at l_0.

Note first that the case where the tree has depth 0 is straightforward: either $\mathcal{P} = \emptyset$ and $\delta = \square$ or $\mathcal{P} = \{l_0\}$ and $\delta = \iota$.

We now consider the inductive case and handle separately each possible definition for the label l_0:

- case of an assignment $l_0 : x = e; l_1$:
  Either $\mathcal{P} = \{l_0 \cdot l_1\}$ or $\mathcal{P}$ is made of paths of the form $l_0 \cdot l_1 \cdot \ldots$. In the former case, $\delta = \delta_{l_0,l_1}$; in the latter case, the induction property ensures that there exists $\delta' \in \mathfrak{S}$, such that $[\![\delta']\!]$ is equal to $\alpha_{p\mathcal{F}[P']}(\mathcal{E})$ where $\mathcal{P}' = \{l_1 \cdot \ldots l_n \mid l_0 \cdot l_1 \cdot \ldots \cdot l_n \in \mathcal{P}\}$, so δ is obtained by composition.

- case of a condition $l_0 : \mathbf{if}(e)\{l_0^t \cdots\}\mathbf{else}\{l_0^f \cdots\}$:
  We let $\mathcal{P}_t = \{l_0^t \cdot \ldots \cdot l_n \mid l_0 \cdot l_0^t \cdot \ldots \cdot l_n \in \mathcal{P}\}$ and $\mathcal{P}_f = \{l_0^f \cdot \ldots \cdot l_n \mid l_0^f \cdot \ldots \cdot l_n \in \mathcal{P}\}$. By induction, we know we can represent the semantics of $\mathcal{P}_t$ (resp. $\mathcal{P}_f$) with $\delta_t \in \mathfrak{S}$ (resp. δ_f). Therefore, we can represent $\{l_0 \cdot l_0^t \cdot \ldots \cdot l_n \in \mathcal{P}\}$ with $\delta_t \oplus \lfloor e \ ?\ \iota \mid \square \rfloor = \lfloor e \ ?\ \delta_t \mid \square \rfloor$; similarly, we get $\lfloor e \ ?\ \square \mid \delta_f \rfloor$ in the case of the false branch. In the end, we get:

$$\alpha_{p\mathcal{F}[\mathcal{P}]}(\mathcal{E}) = \lfloor e \ ?\ \delta_t \mid \delta_f \rfloor$$

- other cases can be handled similarly (with composition of transfer functions and joins for conditions).

This concludes the proof. $\square$

Use in static analysis: Similar symbolic transfer functions were originally introduced by [CL96] as a means to increase the precision of static analyses.

Let us assume that $D^\sharp_{\mathbb{M}}$ defines a Galois-connection (Definition 2.3.1) (we let $\alpha_{\mathbb{M}}$ denote the abstraction function). We write $\delta^\sharp$ for $\alpha_{\mathbb{M}} \circ [\![\delta]\!] \circ \gamma_{\mathbb{M}}$ (most precise abstract transfer function corresponding to δ; an upper approximation of it is usually computed).

If $\zeta = [\![\delta_0]\!] \circ \ldots \circ [\![\delta_n]\!]$ then $\zeta^\sharp \sqsubseteq \delta_0^\sharp \circ \ldots \circ \delta_n^\sharp$ since $\lambda x \cdot x \sqsubseteq \gamma_{\mathbb{M}} \circ \alpha_{\mathbb{M}}$. In general, $\zeta^\sharp \sqsubset \delta_0^\sharp \circ \ldots \circ \delta_n^\sharp$: this strict inequality corresponds to a loss of precision.

For instance, relational abstract domains often handle more precisely complex operations (assignments and guards of complex expressions) when done in one step as is the case for the octagons [Min01] for some linear assignments like $y := \Sigma_i\, a_i \star x_i$ where $a_i \in \mathbb{Z}$. Symbolic transfer functions help in such cases, since they group atomic assignments together and form larger expressions, which the domain may analyze better than the sequence of simple assignments.

Use in program transformations: Symbolic transfer functions allow to express and handle in a computer the denotational semantics along paths and along finite sets of paths (introduced in Section 3.2.2 and Section 3.2.3); this is most useful in order to prove e.g., that a program transformation preserves some abstraction of the standard semantics, as will be done for compilation, in Chapter 9.

3.3 Backward Semantics and Analysis

The previous section introduced denotational semantics as a function from inputs to outputs. However, in some cases, one may be interested in the converse; therefore, we define a backward semantics as well. Furthermore, we extend the abstract interpretation of the denotational semantics.

3.3.1 Backward semantics

Abstraction into backward functions: The backward abstraction is a function mapping any output state to the set of input states, which may lead to it:

Definition 3.3.1. Backward semantics.

The backward abstraction of sets of traces is defined by:

$$(\mathcal{P}(\Sigma), \subseteq) \underset{\alpha_{\overleftarrow{\mathcal{F}}}}{\overset{\gamma_{\overleftarrow{\mathcal{F}}}}{\leftrightarrows}} (\mathbb{M} \to \mathcal{P}(\mathbb{M}), \subseteq)$$

$$\begin{array}{rrcl} \alpha_{\mathcal{F}} : & \mathcal{P}(\Sigma) & \to & (\mathbb{M} \to \mathcal{P}(\mathbb{M})) \\ & \mathcal{E} & \mapsto & \lambda(\rho_0 \in \mathbb{S}).\{\rho_n \mid \langle(\ell_0, \rho_0), \ldots, (\ell_n, \rho_n)\rangle \in \mathcal{E}\} \end{array}$$

(the concretization $\gamma_{\overleftarrow{\mathcal{F}}}$ can be derived from $\alpha_{\overleftarrow{\mathcal{F}}}$ straightforwardly)

This semantics is equivalent to a backward predicate transformer [Dji75].

Obviously, this abstraction is equivalent to the (forward) denotational abstraction introduced in Section 3.2.1, as remarked in [Cou97a]. Indeed, we can turn an "input-to-output" mapping into an "output-to-input" mapping (and vice-versa) by applying the following function, which is defined for any pair of sets (A, B):

$$\begin{array}{rlcl} \mathbf{Inv} : & (A \to \mathcal{P}(B)) & \longrightarrow & (B \to \mathcal{P}(A)) \\ & f & \mapsto & \lambda(b \in B) \cdot \{a \in A \mid f(a) = b\} \end{array}$$

In particular, for any set of traces $\mathcal{E}$, the following properties hold:

$$\alpha_{\mathcal{F}}(\mathcal{E}) = \mathbf{Inv}(\alpha_{\overleftarrow{\mathcal{F}}}(\mathcal{E})) \qquad \alpha_{\overleftarrow{\mathcal{F}}}(\mathcal{E}) = \mathbf{Inv}(\alpha_{\mathcal{F}}(\mathcal{E}))$$

Extension to backward semantics: In Section 3.2, we composed the abstraction to functions with "path" or "from-to" abstractions, so as to choose the granularity of the backward semantics. The same step should also be done in the case of the backward semantics. In particular, we can define the backward semantics between two control states. For each pair $(\ell_\vdash, \ell_\dashv) \in \mathbb{S}^2$:

$$\alpha_{t\overleftarrow{\mathcal{F}}[\ell_\vdash, \ell_\dashv]}(\mathcal{E}) = \alpha_{\mathcal{F}}(\{\langle(\ell_0, \rho_0), \ldots, (\ell_n, \rho_n)\rangle \in \mathcal{E} \mid \ell_0 = \ell_\vdash \wedge \ell_n = \ell_\dashv\})$$

3.3.2 Backward static analysis

The approximation of the co-reachable states from a set of *final* states proceed in a similar way as the forward analysis described in Figure 3.3.

We write $\overleftarrow{[\![s]\!]}^\sharp : D^\sharp_{\mathbb{M}} \to D^\sharp_{\mathbb{M}}$ for a backward semantics for statements. Such a function should be sound in the usual way: it should compute an over-approximation of the set of input states which may lead to some output state.

Such a backward interpreter is displayed in Figure 3.6 This interpreter is sound:

statement s	abstract semantics
$x := e;$	$\overleftarrow{[\![s]\!]^\sharp} : d \mapsto \overleftarrow{\mathit{assign}}(x, e, d)$
$\mathbf{if}(e)\{s_0\}\mathbf{else}\{s_1\}$	$\overleftarrow{[\![s]\!]^\sharp} : d \mapsto \mathit{guard}(e, \mathbf{true}, \overleftarrow{[\![s_0]\!]^\sharp}(d)) \sqcup \mathit{guard}(e, \mathbf{false}, \overleftarrow{[\![s_1]\!]^\sharp}(d))$
$\mathbf{while}(e)\{s\}$	$[\![s]\!]^\sharp : d \mapsto \mathbf{lfp}^\sharp_{\mathit{guard}(e,\mathbf{false},d)} F^\sharp$ where $F^\sharp : D^\sharp_{\mathbb{M}} \to D^\sharp_{\mathbb{M}}$, $d_0 \mapsto \mathit{guard}(e, \mathbf{true}, \overleftarrow{[\![s]\!]^\sharp}(d_0))$ and $\mathbf{lfp}^\sharp$ computes an abstract post-fixpoint
$\mathbf{input}(x \in V);$	$\overleftarrow{[\![s]\!]^\sharp} : d \mapsto \mathit{forget}(x, d)$
$\mathbf{assert}(e);$	$\overleftarrow{[\![s]\!]^\sharp} : d \mapsto \mathit{guard}(e, \mathbf{true}, d)$

Figure 3.6: Backward abstract interpreter

Theorem 3.3.1. Soundness of the backward abstract interpreter.
Let $\ell_\vdash : s : \ell_\dashv$ be a program, $\rho_\vdash, \rho_\dashv \in \mathbb{M}$, and $d \in D^\sharp_{\mathbb{M}}$. Then,

$$\left.\begin{array}{l} \rho_\dashv \in \gamma_{\mathbb{M}}(d) \\ \rho' \in \alpha_{\overleftarrow{t\mathscr{F}}[\ell_\vdash,\ell_\dashv]}([\![s]\!])(\rho_\dashv) \end{array}\right\} \Longrightarrow \rho_\vdash \in \gamma_{\mathbb{M}}(\overleftarrow{[\![s]\!]^\sharp}(d))$$

A number of refinements to this simple analysis could be implemented. In particular, we may use the forward analysis to refine the resulting invariants, by doing *local iterations* [Gra92].

3.4 Projection Abstractions

In some cases, we may wish to forget only part of the information about the history of the execution of programs, while keeping other relevant parts of the history of executions. Therefore, we propose some families of "*projection*" abstractions, which allow to forget about part of the execution of programs, by fixing some "granularity" for the observation of states and projecting states in traces according to this observation.

Along this section, we consider a very simple program transformation as an example to illustrate the various definitions: constant propagation [Kil73] with dead code elimination [WM94]. We derive the semantics of the target program by applying some projections to the semantics of the source program.

In this section, we focus on the following running example:

$$\begin{array}{ll}
\ell_0: & \mathrm{int}\, i = 4; \\
\ell_1: & \mathrm{int}\, j = 5; \\
\ell_2: & \mathrm{int}\, k, l; \\
\ell_3: & \mathbf{if}(i < j)\{ \\
\ell_4: & \quad k = i + j; \\
\ell_5: & \}\,\mathbf{else}\,\{ \\
\ell_6: & \quad k = k - j; \\
\ell_7: & \} \\
\ell_8: & l = l + k; \\
\ell_9: & \ldots
\end{array}$$

(a) Source program P_s

$$\begin{array}{ll}
\ell_2: & \mathrm{int}\, l; \\
\ell_8: & l = l + 9; \\
\ell_9: & \ldots
\end{array}$$

(b) Transformed program P_r

Figure 3.7: Constant propagation

Definition 3.4.1. Constant propagation and dead code elimination.

Let us consider the program in Figure 3.7(a). Constant propagation reveals that i, j are constant, and the condition $i < j$ evaluates to **true***; hence, k is also constant. As a result most statements (and variables) can be removed: the program in Figure 3.7(b) produces the same result, as far as l is concerned.*

3.4.1 Variable projection

The first kind of projection we consider proceeds by abstracting away some *memory locations*. More precisely, if $\mathbb{X}$ denotes the set of memory locations, then, we let $\overline{\mathbb{X}} \subseteq \mathbb{X}$ be a *restricted set* of memory locations, collecting the memory locations, which still appear in the transformed program. We write $\overline{\mathbb{M}}$ for $\overline{\mathbb{X}} \to \mathbb{V}$, and we also let $\overline{\Sigma}$ denote the set of traces over $\overline{\mathbb{M}}$. We let the store projection operator $\Pi^{\text{store}}_{\overline{\mathbb{X}}}$ be defined by:

$$\begin{array}{rrcl}
\Pi^{\text{store}}_{\overline{\mathbb{X}}}: & \mathbb{M} & \longrightarrow & \overline{\mathbb{M}} \\
& \rho & \longmapsto & \lambda(x \in \overline{\mathbb{X}}) \cdot \rho(x)
\end{array}$$

This function can be lifted into a projection of traces in a straightforward way. It allows to define an observation of the semantics of programs, which takes into account only part of the variables of the program.

In the case of constant propagation, a variable which is proved constant by the initial analysis can be removed from the program; therefore, it should not be included in $\overline{\mathbb{X}}$.

Definition 3.4.2. Constant propagation and variable removal.

For instance, in the example in Figure 3.7, the variables i, j, k are proved constant and propagated; hence, they should be removed: $\overline{\mathbb{X}} = \{l\}$.

3.4.2 Control states projection

The second kind of projection we consider abstracts away some *control states.* More precisely, if $\mathbb{L}$ denotes the set of control states, then, we let $\overline{\mathbb{L}} \subseteq \mathbb{L}$ denote the set of *restricted set* of control states, which still appear in the transformed program. We let the trace projection operator $\Pi^{\text{trace}}_{\overline{\mathbb{L}}}$ be defined by:

$$\begin{array}{rlcl} \Pi^{\text{trace}}_{\overline{\mathbb{L}}}: & \Sigma & \rightarrow & \overline{\Sigma} \\ & \langle(l_0, \rho_0), \ldots, (l_n, \rho_n)\rangle & \mapsto & \langle(l_{i_0}, \rho_{i_0}), \ldots, (l_{i_k}, \rho_{i_k})\rangle \\ & & & \text{where} \left\{ \begin{array}{l} i_0 < i_1 < \ldots < i_n \\ \{i_j \mid j \in (\!|0, k|\!)\} = \{i \in (\!|0, n|\!) \mid l_i \in \mathbb{L}\} \end{array} \right. \end{array}$$

Intuitively, it erases any state corresponding to a control state not in $\overline{\mathbb{L}}$. In case we added an error state Ω, it should also be preserved by $\Pi^{\text{trace}}_{\overline{\mathbb{L}}}$.

Definition 3.4.3. Control states removal.

In the case of constant propagation and dead code removal we should abstract away:
- *control states corresponding to unreachable states (though, this projection does not change the semantics, since it erases states which are not reachable): this is the case of l_6 and l_7 in the example;*
- *control states corresponding to propagated, constant assignments, as is the case for l_4, l_5 in the example (note that the conditional at l_3 can be removed as well).*

3.4.3 General case

In practice, both control states and memory location projections need to be used in the same time. For instance, the example displayed in Figure 3.7 requires both the removal of some control states and of some memory locations. Therefore, we use the following notations:
- $\Pi^{\text{store}}_{\overline{\mathbb{X}}}$ and $\Pi^{\text{state}}_{\overline{\mathbb{X}}}$ were defined in Section 3.4.1;
- $\Pi^{\text{trace}}_{\overline{\mathbb{X}},\overline{\mathbb{L}}}$ carries out the control states projection mentioned in Section 3.4.2 and applies $\Pi^{\text{state}}_{\overline{\mathbb{X}}}$ to the remaining states.

The projection abstraction of sets of traces is defined by:

Definition 3.4.1. Projection abstraction.

Let the functions $\alpha_{\Pi\langle\overline{\mathbb{X}},\overline{\mathbb{L}}\rangle}$ and $\gamma_{\Pi\langle\overline{\mathbb{X}},\overline{\mathbb{L}}\rangle}$ by defined by:

$$\begin{array}{rlcl} \alpha_{\Pi\langle\overline{\mathbb{X}},\overline{\mathbb{L}}\rangle}: & \mathcal{P}(\Sigma) & \rightarrow & \mathcal{P}(\Sigma) \\ & \mathcal{E} & \mapsto & \{\Pi^{\text{trace}}_{\overline{\mathbb{X}},\overline{\mathbb{L}}}(\sigma) \mid \sigma \in \mathcal{E}\} \end{array}$$

$$\begin{array}{rlcl} \gamma_{\Pi\langle\overline{\mathbb{X}},\overline{\mathbb{L}}\rangle}: & \mathcal{P}(\Sigma) & \rightarrow & \mathcal{P}(\Sigma) \\ & \mathcal{E} & \mapsto & \{\sigma \in \Sigma \mid \Pi^{\text{trace}}_{\overline{\mathbb{X}},\overline{\mathbb{L}}}(\sigma) \in \mathcal{E}\} \end{array}$$

Then, there is a Galois connection $(\mathcal{P}(\Sigma), \subseteq) \xleftrightarrow[\alpha_{\Pi\langle\overline{\mathbb{X}},\overline{\mathbb{L}}\rangle}]{\gamma_{\Pi\langle\overline{\mathbb{X}},\overline{\mathbb{L}}\rangle}} (\mathcal{P}(\Sigma), \subseteq)$.

Definition 3.4.4. Constant propagation and dead code elimination.

In the example given in Figure 3.7, the following restricted sets shall be used:

- $\overline{\mathbb{X}} = \{l\}$ *(*$\mathbb{X} = \{i, j, k, l\}$*);*
- $\overline{\mathbb{L}} = \{l_2, l_8, l_9\}$ *(*$\mathbb{L} = \{l_i \mid i \in [\![0, 9]\!]\}$*).*

The correctness of the constant propagation and dead code removal transformation can be described by the abstraction relation:

$$[\![P_r]\!] = \alpha_{\Pi\langle\overline{\mathbb{X}},\overline{\mathbb{L}}\rangle}([\![P_s]\!])$$

Intuitively, all traces of the transformed system are obtained from the traces of the original system by removing all control states and memory locations, but those in $\overline{\mathbb{X}}, \overline{\mathbb{L}}$*.*

This relation is a particular case of the scheme introduced in Section 2.3.4. The formalization of program transformations often requires this kind of abstraction, e.g. when some parts of the source program are deleted. A similar approach shall be used in the formalization of other program transformations, such as slicing and compilation.

3.4.4 Fixpoint-based definition

A fixpoint based definition is always very convenient in order to establish semantic properties of programs (e.g., static analysis); therefore, we attempt to give some fixpoint definition for the projection of the semantics of a program P.

We consider the case of control state projection first (i.e., we assume that $\overline{\mathbb{X}} = \mathbb{X}$):

Lemma 3.4.1. Fixpoint definition.

Let P *be a program, and* $\overline{\mathbb{L}} \subseteq \mathbb{L}$*.*
Then, $\alpha_{\Pi\langle\overline{\mathbb{L}}\rangle}([\![P]\!])$ *is strongly closed.*
Moreover, $\alpha_{\Pi\langle\overline{\mathbb{L}}\rangle}([\![P]\!])$ *writes down as a least fixpoint: there exists* $F : \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$*, such that:*

$$\alpha_{\Pi\langle\overline{\mathbb{L}}\rangle}([\![P]\!]) = \mathbf{lfp}_{\emptyset} F$$

Proof.

We write $\mathcal{E}$ for $\alpha_{\Pi\langle\overline{\mathbb{L}}\rangle}([\![P]\!])$.
We start with the proof of strong closeness. Let $\sigma'_0, \sigma'_1 \in \mathcal{E}$ such that $\sigma'_0 \frown \sigma'_1$ is defined. Therefore, we can write down $\sigma'_0 = \langle \ldots, (l, \rho)\rangle$ and $\sigma'_1 = \langle (l, \rho)\rangle$ (the concatenation of σ'_0 and σ'_1 exists; hence, the last state in σ'_0 is the same as the first state in σ'_1). Moreover, there exist $\sigma_0, \sigma_1 \in [\![P]\!]$ such that $\alpha_{\Pi\langle\overline{\mathbb{L}}\rangle}(\sigma_0) = \sigma'_0$ and $\alpha_{\Pi\langle\overline{\mathbb{L}}\rangle}(\sigma_1) = \sigma'_1$ and we can choose σ_0, σ_1 such that the last state of σ_0 is (l, ρ), and the same for the first state of σ_1. As a

consequence, $\sigma_0 \frown \sigma_1$ exists. Last, we can prove easily that $\alpha_{\Pi\langle\overline{\mathbb{L}}\rangle}(\sigma_0 \frown \sigma_1) = \sigma'_0 \frown \sigma'_1$, so $\sigma'_0 \frown \sigma'_1 \in \mathcal{E}$.
The converse implication is straightforward (i.e., $\mathcal{E}$ is closed: if $\sigma'_0 \frown \sigma'_1 \in \mathcal{E}$, then $\sigma'_0, \sigma'_1 \in \mathcal{E}$).

Last, the fixpoint definition follows from Theorem 3.2.3. □

In the case of store projection, such a result is not automatic. Indeed, the property $(\sigma'_0, \sigma'_1 \in \mathcal{E}) \Longrightarrow \sigma'_0 \frown \sigma'_1 \in \mathcal{E}$ does not hold, because the last state of σ_0 and the first state of σ_1 (where σ_0, σ_1 are defined as above) may not be equal: in fact the assumption only guarantees the equality of the abstractions of these states to $\overline{\mathbb{X}}$.

3.5 Hierarchies of abstractions

We presented several semantics in this chapter, and stated abstraction relations between some of them. For instance, we described a common static analysis framework as an abstraction of trace semantics in Section 3.1.

This approach can be used in a systematic way, for defining, comparing, and integrating different semantics in *hierarchies of abstractions*. In particular, [Cou97a] relates various abstraction of trace semantics in a hierarchy of abstractions. Other authors applied this approach to other families of semantics: for instance, [GM03] extended the standard traces into transfinite traces (i.e., sequences of elements indexed with ordinal numbers) and derive other kinds of semantics as abstractions.

In the following, we use the common abstractions recalled in this chapter and define new abstractions, so that we could relate them in hierarchies as well, even though we do not present it this way. For instance, the formalization of the invariant translation technique (Chapter 10) will require a common abstraction of the static analysis of Section 3.1 and of the semantic projection of Section 3.4 to be defined.

Part II

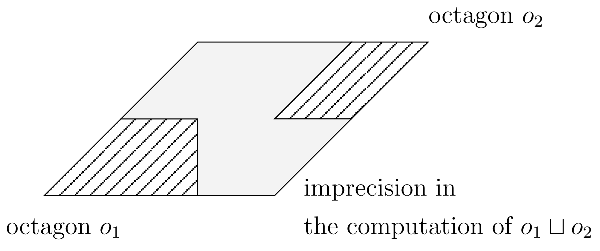
Trace Partitioning

Chapter 4

A Framework for Partitioning Traces

As mentioned in Section 3.1, generic abstract interpreters can be defined, which compute an over-approximation of the reachable states, and which accept an abstract domain for representing sets of stores as a parameter. This domain expresses various kinds of constraints among variables. Most of the domains cited in Section 3.1.3 cannot express any non trivial disjunction, which might be necessary for some property (such as the absence of runtime errors) to be proved. Indeed, many commonly used abstract domains like intervals, octagons, polyhedra only express convex constraints; therefore the abstract join operation incurs a loss of precision as depicted in the Figure below, which may not allow the property of interest to be proved successfully.

Furthermore, some disjunctions might be necessary, that do not involve only program variables but more complex properties, such as (an abstraction of) the history of execution, therefore it is desirable to provide abstractions, which allow to express disjunctions and to take the properties of program executions into account. The purpose of this Part of the thesis is to introduce families of abstractions of traces, which allow to express such constraints. These abstractions perform a partitioning of the set of traces, based on the observation of the history of executions.



This chapter aims at introducing a general framework for control-based trace partitioning [MR05]. Section 4.1 reviews common cases of partitioning and motivates the need for further abstractions to be developed. Section 4.2 introduces and formalizes the core trace partitioning framework. Section 4.3 focuses on the application of this framework to static analysis, using *static* or *dynamic* partitions.

4.1 Partitioned Systems

4.1.1 Partitioning control states

First of all, we underline that partitioning the reachable states with the control states is a rather common approach in static analysis. Later, we generalize drastically this technique.

Non procedural case: Indeed, the analysis proposed in Section 3.1 relies on this kind of partitioning. The abstraction of sets of traces can be seen as a two steps abstraction:

1. abstraction of **traces into states**, with *partitioning*:

$$(\mathcal{P}(\Sigma), \subseteq) \underset{\alpha_{\mathbb{P}(\mathbb{L})}}{\overset{\gamma_{\mathbb{P}(\mathbb{L})}}{\leftrightarrows}} (\mathbb{L} \to \mathcal{P}(\mathbb{M}), \subseteq)$$

$$\begin{array}{rlll} \alpha_{\mathbb{P}(\mathbb{L})} : & \mathcal{P}(\Sigma) & \to & (\mathbb{L} \to \mathcal{P}(\mathbb{S})) \\ & \mathcal{E} & \mapsto & \lambda(\ell \in \mathbb{L}) \cdot \{\rho \mid \langle \ldots, (\ell, \rho), \ldots \rangle \in \mathcal{E}\} \end{array}$$

Whenever the concretization function is defined straightforwardly from the abstraction function, we provide the abstraction function only: in a complete lattice, any monotone abstraction function defines a unique concretization [CC77].

2. abstraction of **sets of states**, defined by the concretization function $\gamma_{\mathbb{M}} : D^{\sharp}_{\mathbb{M}} \to \mathcal{P}(\mathbb{M})$.

Note that the abstraction in step 1 collects the stores in the end of traces; this is equivalent to collecting all stores in traces since we consider *closed* sets of traces (Section 3.2.4): if σ is an execution of a program P, then any prefix of σ is also an execution of P.

The first step includes a partitioning in the sense of [CC92a, §4.2.3.2]. Indeed, it amounts to partitioning the set of sets of states using the partition $\{\{(\ell, \rho) \mid \rho \in \mathbb{M}\} \mid \ell \in \mathbb{L}\}$; the resulting domain is in bijection with $\mathbb{L} \to \mathcal{P}(\mathbb{M})$.

Procedural case: In case the language features procedures, similar abstractions are usually implemented.

We consider the procedural extension introduced in Section 2.2.4. When designing an analysis for such a procedural language, one faces the problem of deciding how to replace the abstraction mentioned in step 1 above. Among the possible choices, we can cite [SP81]:

- the **full abstraction of the stack:** we may abstract away the stack and keep only the control states (analysis insensitive to the calling context):

$$\begin{array}{rlll} \alpha_{\mathbb{P}(\mathbb{L})} : & \mathcal{P}(\Sigma) & \to & (\mathbb{L} \to \mathcal{P}(\mathbb{M})) \\ & \mathcal{E} & \mapsto & \lambda(l \in \mathbb{L}) \cdot \{\rho \mid \exists \kappa \in \mathbb{k},\ \exists \langle \ldots, (\kappa, \ell, \rho), \ldots \rangle \in \mathcal{E}\} \end{array}$$

- the **partitioning with the stack:** we may keep the stack, i.e. abstract traces into functions mapping pairs made of a stack and a control state into a set of memory states (analysis completely sensitive to the calling context):

$$\begin{array}{rlcl} \alpha_{\mathbb{P}(\mathbb{L})} : & \mathcal{P}(\Sigma) & \rightarrow & ((\mathbb{k} \times \mathbb{L}) \rightarrow \mathcal{P}(\mathbb{S})) \\ & \mathcal{E} & \mapsto & \lambda((\kappa, \ell) \in (\mathbb{k} \times \mathbb{L})) \cdot \{\rho \mid \langle \ldots, (\ell, \rho), \ldots \rangle \in \mathcal{E}\} \end{array}$$

This approach amounts to inlining functions; it works only in the case of non-recursive function calls (the stack may grow infinite in the case of recursive calls). At the time this thesis is written, this is the technique implemented in ASTRÉE.

Many intermediate abstractions exist, which allow to retain a good level of precision in some cases and abstract long sequences of calls (the main such technique is k-limiting).

Another approach to the analysis of procedural programs is to modelize the effect of each function (intra-procedural phase) and then, to perform a global iteration [RHS95]. This technique relies on the resolution of the reachability along "interprocedural realizable paths", which is also based on some abstraction of the stack (this method was also used in slicing [HRB88]).

4.1.2 Partitioning memory states

Another interesting approach to partitioning consists in partitioning the set of memory states. Let us consider the program on Figure 4.1(a), computing the absolute value of x. We assume that the variables x, sgn have mathematical integer values (we do not consider machine integers, modular arithmetics or possible overflows into account here). Then, this program is safe in the sense that it never crashes whatever the initial value for x (in the case of 32-bits machine integers, it would not work as expected for $x = -2^{32}$, which is the reason for the above assumption). However, if we analyze it with the domain of intervals (using the abstract interpreter introduced in Section 3.1), we would find:

- $sgn = -1$ at ℓ_2;
- $sgn = 1$ at ℓ_4;
- $sgn \in [-1, -1] \sqcup [1, 1]$ at ℓ_5, i.e. $sgn \in [-1, 1]$.

As a consequence, the analysis would report a possible division by 0 at point ℓ_5, since $0 \in [-1, 1]$. We note that this stems from an imprecision due to the abstract join computed at the exit of the conditional. Furthermore, the analyzer would not prove that $y \geq 0$ at ℓ_6, due to the lack of relation between sgn and the sign of x in the abstract environment.

A first possible refinement relies on disjunctive completion [CC79], i.e., the possible values for a variable are abstracted into the union of a set of intervals. An important drawback of disjunctive completion is its cost: when applied to a finite domain of cardinal n, it produces a domain of 2^n elements, with chains of length $n + 1$. Moreover, the design of a widening for the domains obtained by disjunctive completion is a non-trivial issue; in particular, a good widening operator should decide which elements of a partition to merge or to widen.

A second solution to these issues is to refine the abstract domain, so as to express a *relation* between x and sgn. For instance, we would get the following constraint, at point

```
      int $x, sgn$;
$l_0$ **if**$(x < 0)\{$
$l_1$     $sgn = -1$;
$l_2$ $\}$**else**$\{$
$l_3$     $sgn = 1$;
$l_4$ $\}$
$l_5$ $y = x/sgn$;
$l_6$ $\ldots$
```

(a) Absolute value

```
      int $x, y$;
$l_0$ $n = 0$;
$l_1$ $y = 0$;
$l_2$ **while**(**true**)$\{$
$l_3$     $y = y + (-1)^n * 5$;
$l_4$     $n = n + 1$;
$l_5$ $\}$
$l_6$ $\ldots$
```

(b) Alternating iterations

```
      int $i$;
      float $x, y$;
      $x$ is assumed to be in a range $[0, n]$
$l_0$ int $i = 0$;
$l_1$ $i = \mathbf{cast}_{\text{float}\to\text{int}} x$;
$l_2$ $y = ty[i] + (x - \mathbf{cast}_{\text{int}\to\text{float}} i) *$
          $(ty[i+1] - ty[i])$
$l_3$ $\ldots$
```

(c) Interpolation

Figure 4.1: Examples

ℓ_5:

$$\begin{cases} x < 0 & \Rightarrow & sgn = -1 \\ x \geq 0 & \Rightarrow & sgn = 1 \end{cases}$$

Such an abstraction would be very costly if applied exhaustively, to any variable (especially if the program to analyze contains thousands of variables, as is the case of the applications mentioned in Section 5.1.1), therefore a strategy should be used in order to determine which relations may be useful to improve the precision of the result. However, the choice of the predicate which should guide the partitioning (i.e., $x < 0$ in the above example) may not always be obvious.

4.1.3 Other partitioning criteria

The two previous subsections presented partitioning abstractions which are necessary in order to produce relevant results (partitioning with control states) and precise results (partitioning with the values of some variables). However, these techniques are not completely satisfactory.

- First of all, we noted in Section 4.1.2 that the design of partitioning numerical abstract domains is not easy, due to the cost and the need for choosing accurately what relation to use for the partitioning and to issues in the design of efficient widening operators.
- Secondly, this kind of partitioning will not allow to express all the constraints we might be interested in. For instance, in the program displayed in Figure 4.1(b), a naive interval analysis will not bound the value of y. However, the values of y are cyclic: at ℓ_3, after an odd number of iterations in the loop, $y = 0$ and, after an even number of iterations, $y = 5$, so that y is bounded.
  Consequently, one would succeed in proving the property of interest by performing a partitioning of the memory states based on the parity of the variable i. Again, this solution presents a major drawback: the analyzer would have to *choose* what predicate to use in order to perform the right partitioning; in particular, it should choose the variable i, possibly among thousands of other variables.
  However, the above property is clearly based on a case analysis on an abstraction of the history of the execution of the program (case analysis on the parity of the number of iterations).
  In this example, the goal for the partitioning is to prove the safety of the program; however, it may be to express some properties of programs. For instance, we may want to prove that some property holds for certain iteration numbers or to show that if some property holds at iteration n, then some other property holds at iteration $n - 1$ or $n + 1$. All these cases involve similar disjunctions based on the history of the control flow: most of the time, the disjunctions of interest can be read in the control flow.
- In the program in Figure 4.1(a), the disjunction which is needed in order to prove the safety of the program also amounts to a case analysis on the history of the

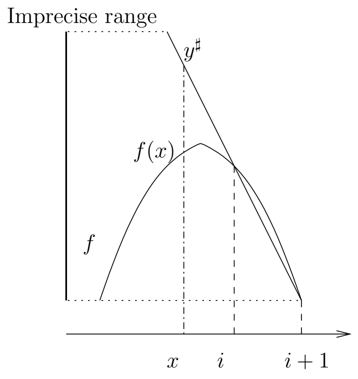


Figure 4.2: Analysis of an interpolation and imprecision

execution. Indeed, at point ℓ_5:

  - if the execution flowed through the **true** branch, then $sgn = -1$;
  - if the execution flowed through the **false** branch, then $sgn = 1$.

- Other kinds of disjunctions, which do not naturally follow from tests can be expressed easily in this framework.
  For instance, let us consider the case of the interpolation function in Figure 4.1(c), which aims at approximating a function $f : \mathbb{R} \rightarrow \mathbb{R}$ (for instance, $f = \sin, \cos \ldots$), using a discretization, and an approximation with floating point numbers and linear interpolations. Basically, the array ty represents the value of f for integer values: $ty[i]$ approximates the value of $f(i)$.
  When analyzing this program with, e.g. the domain of intervals, a range is computed for i at ℓ_1, and then, the assignment at ℓ_2 is performed: since the memory locations corresponding to the array lookups depend on i, the analyzer should consider any value in the range known for i at ℓ_2 and compute the join of all the results. Not only this join would incur a loss of precision, but also, the application of the formula on the right hand side for a value of x and a value of i such that $i \neq \lfloor x \rfloor$ may lead to very imprecise results We can see this imprecision in Figure 4.2, where x is not in the range $[i, i+1]$ and the abstract computation generates a very imprecise result $y^\sharp$, compared to the concrete result $f(x)$:
  This issue can be solved either by a partitioning by the value of i inside the domain $D^\sharp_{\mathbb{M}}$ or by a partitioning of the traces by the value of i at point ℓ_1. The advantage of the latter approach is that the partitions do not need to be recomputed if i is assigned again at some point. Indeed, the partitioning is guided by the value of i *as a result of the statement at* ℓ_1, and *not* by the value of i at any time.

Last, a crucial point is that the partitions should not be global: making them local should help in reducing the cost of partitioning to a minimum.

4.2 Control Partitioning of Transition Systems

4.2.1 Partitions and coverings

We first set up the notions of *partitioned set* and *partitioned system.*

Partitioning function: A *covering* of a set F is a family of subsets of F, such that any element of F belongs to some element of the family. A *partition* is a covering such that any two distinct elements of the family are disjoint; in particular, for any element $x \in F$, there exists a unique element A of the partition such $x \in A$. In the following, we need to index the elements of coverings (resp. partitions); hence, the following definition resorts to functions, defined on a set of *indexes.*

Definition 4.2.1. Partitioned set.

Let E, F be two sets, and $\delta : E \to \mathcal{P}(F)$. Then:

- *δ is a* covering *of F if and only if:*

$$\forall x \in E,\ \delta(x) \neq \emptyset$$

and,

$$F = \bigcup_{x \in E} \delta(x)$$

- *δ is a* partition *of F if and only if it is a covering and:*

$$\forall x, y \in E,\ x \neq y \Longrightarrow \delta(x) \cap \delta(y) = \emptyset$$

We note that a covering (resp. partitioning) δ of F defines an abstraction of $(\mathcal{P}(F), \subseteq)$:

Lemma 4.2.1. Partitioning abstraction.

Let $\alpha_{\mathfrak{P}(\delta)}$ and $\gamma_{\mathfrak{P}(\delta)}$ be defined by:

$$\begin{array}{rlll} \alpha_{\mathfrak{P}(\delta)} : & \mathcal{P}(F) & \to & (E \to \mathcal{P}(F)) \\ & \mathcal{E} & \mapsto & \lambda(x \in E) \cdot \mathcal{E} \cap \delta(x) \\ \\ \gamma_{\mathfrak{P}(\delta)} : & (E \to \mathcal{P}(F)) & \to & \mathcal{P}(F) \\ & \phi & \mapsto & \bigcup_{x \in E} \phi(x) \end{array}$$

Then, if δ is a covering, we have a Galois-connection $(\mathcal{P}(F), \subseteq) \xleftrightarrows[\alpha_{\mathfrak{P}(\delta)}]{\gamma_{\mathfrak{P}(\delta)}} (E \to \mathcal{P}(F), \subseteq$), and $\alpha_{\mathfrak{P}(\delta)}$ is into (Galois injection).
Moreover, if δ is a partition, then $\alpha_{\mathfrak{P}(\delta)}$ is one-to-one (Galois bijection).

Proof.
Straightforward application of the definition of coverings. □

Definition 4.2.1 would allow to set up very general notions of trace partitioning. In particular, the partitioning of traces using the control state of the traces (Section 4.1.1) of the last state fits in this framework (with $E = \mathbb{L}$); the case of calling stacks is similar (with either $E \equiv \mathbb{L}$, or $E = \mathbb{k} \times \mathbb{L}$, or other partitions). We may even design some weaker partitions: for instance, we may decide to merge together the state corresponding to several distinct control states (with E a partition of $\mathbb{L}$). However, we wish to derive the partitions from the history of executions; therefore the following paragraph introduces the notion of *partitioned system*.

Partitioning transitions: In the following, we assume that a program P is given, and defined by $(\mathbb{L}, \mathbb{S}^{\mathrm{i}}, \rightarrow)$. We consider partitions finer than the partition defined by $E = \mathbb{L}$ only. More precisely, we let $\mathbb{T}$ be a set of tokens, and $\mathfrak{T} = \mathcal{P}(\mathbb{T})$.

We define *extended transition systems* as transition systems over the sets of labels extended with a set of tokens $T \subseteq \mathbb{T}$; it is basically defined by T and by extensions of the set of initial states and of the transition relation. Such a system P_0 is a covering of P_1 if and only if it simulates the transitions of P_1; moreover, P_0 is a partition if and only if any transition in P_1 is simulated by exactly *one* transition in P_0 (and the same for the initial states). System P_0 is complete in case it does not add any fictitious transition, when compared to P_1. Intuitively, a complete partition or covering P_0 shall describe the same set of traces as P_1, up-to some information added in the control states. The main difference between a covering and a partition is that the covering may not ensure the unicity of the counterpart of the traces of the initial program.

The extra information embedded in the control structure of the extended system will be the basis of the partitioning abstraction. The notions of covering, partitioning and complete systems are formalized in the following definition.

Definition 4.2.2. Partitioned system.

Let $T \in \mathfrak{T}$. We write $\mathbb{L}_T$ for the set of partitioned control states $\mathbb{L} \times T$, *$\mathbb{S}_T$ for the set of* partitioned states $\mathbb{L}_T \times \mathbb{M}$, *and $\mathbb{S}^{\mathrm{i}}_T \subseteq \mathbb{S}_T$ for a set of* partitioned initial states, *and $\rightarrow_T$ for a transition relation among partitioned states. An* extended system *is defined by the data of a tuple $(T, \mathbb{S}^{\mathrm{i}}_T, \rightarrow_T)$. Last, Σ_T denotes the set of traces made of states in $\mathbb{S}_T$. For all $T, T' \in \mathfrak{T}$ and $\tau : T \rightarrow T'$, we define the* forget functions *for control states, for states and for traces as follows:*

$$\begin{array}{rlcl}
\pi^{\mathbb{L}}_{\tau} : & \mathbb{L}_T & \rightarrow & \mathbb{L}_{T'} \\
& (\ell, t) & \mapsto & (\ell, \tau(t)) \\
\pi^{\mathbb{S}}_{\tau} : & \mathbb{S}_T & \rightarrow & \mathbb{S}_{T'} \\
& ((\ell, t), \rho) & \mapsto & (\pi^{\mathbb{L}}_{\tau}(\ell, t), \rho) \\
\pi^{\Sigma}_{\tau} : & \Sigma_T & \rightarrow & \Sigma_{T'} \\
& \langle s_0, \ldots, s_n \rangle & \mapsto & \langle \pi^{\mathbb{S}}_{\tau}(s_0), \ldots, \pi^{\mathbb{S}}_{\tau}(s_n) \rangle
\end{array}$$

$$\begin{tikzcd}
& (\ell_0, t) \arrow[dl] \arrow[dr] & \\
(\ell_1, t) \arrow[dr] & & (\ell_2, t) \arrow[dl] \\
& (\ell_3, t) \arrow[d] & \\
& (\ell_4, t) &
\end{tikzcd}$$

(a) System P_0

$$\begin{tikzcd}
& (\ell_0, t_0) \arrow[dl] \arrow[dr] & \\
(\ell_1, t_1) \arrow[d] & & (\ell_2, t_2) \arrow[d] \\
(\ell_3, t_1) \arrow[dr] & & (\ell_3, t_2) \arrow[dl] \\
& (\ell_4, t_0) &
\end{tikzcd}$$

(b) System P_1

$$\begin{tikzcd}
& (\ell_0, t_0) \arrow[dl] \arrow[dr] & \\
(\ell_1, t_1) \arrow[d] & & (\ell_2, t_2) \arrow[d] \\
(\ell_3, t_1) \arrow[d] & & (\ell_3, t_2) \arrow[d] \\
(\ell_4, t_1) & & (\ell_4, t_2)
\end{tikzcd}$$

(c) System P_2

Figure 4.3: Partitioned systems

We consider the extended systems $P_T = (\mathbb{L}_T, \mathbb{L}^{\mathrm{i}}_T, \rightarrow_T)$ and $P_{T'} = (\mathbb{L}_{T'}, \mathbb{L}^{\mathrm{i}}_{T'}, \rightarrow_{T'})$, and the function $\tau : T \rightarrow T'$.

1. *P_T is a τ-*covering *of $P_{T'}$ if and only if:*
   - $\mathbb{S}^{\mathrm{i}}_{T'} \subseteq \pi^{\mathbb{S}}_{\tau}(\mathbb{S}^{\mathrm{i}}_T)$
   - $\forall s_0 \in \mathbb{S}_T,\ s'_1 \in \mathbb{S}_{T'},\ \pi^{\mathbb{S}}_{\tau}(s_0) \rightarrow_{T'} s'_1 \Longrightarrow \exists s_1 \in \mathbb{S}_T,\ \begin{cases} s'_1 = \pi^{\mathbb{S}}_{\tau}(s_1) \\ s_0 \rightarrow_T s_1 \end{cases}$
2. *P_T is a τ-*partition *of $P_{T'}$ if and only if:*
   - $\forall s' \in \mathbb{S}^{\mathrm{i}}_{T'},\ \exists! s \in \mathbb{S}^{\mathrm{i}}_T,\ s' = \pi^{\mathbb{S}}_{\tau}(s)$
   - $\forall s_0 \in \mathbb{S}_T,\ s'_1 \in \mathbb{S}_{T'},\ \pi^{\mathbb{S}}_{\tau}(s_0) \rightarrow_{T'} s'_1 \Longrightarrow \exists! s_1 \in \mathbb{S}_T,\ \begin{cases} s'_1 = \pi^{\mathbb{S}}_{\tau}(s_1) \\ s_0 \rightarrow_T s_1 \end{cases}$
3. *P_T is τ-*complete *with respect to $P_{T'}$ if and only if:*
   - $\forall s \in \mathbb{S}^{\mathrm{i}}_T,\ \pi^{\mathbb{S}}_{\tau}(s) \in \mathbb{S}^{\mathrm{i}}_{T'}$
   - $\forall s_0, s_1 \in \mathbb{S}_T,\ s_0 \rightarrow_T s_1 \Longrightarrow \pi^{\mathbb{S}}_{\tau}(s_0) \rightarrow_{T'} \pi^{\mathbb{S}}_{\tau}(s_1)$

The notions of "complete covering" or "complete partition" are derived from the above definition as well.

Definition 4.2.1. Partitioned systems.

We make the assumption that $\mathbb{M}$ is a singleton here, so that transitions relations are mere relations among control states. Let us consider the two extended systems P_0 and P_1, displayed respectively in Figure 4.3(a) and in Figure 4.3(b).

- *the original system represents a program with a conditional statement followed by one statement (each branch of the conditional contains exactly one statement);*
- *P_0 is isomorphic to the original system; it corresponds to $T_0 = \{t\}$*
- *P_1 is an extended system defined by $T_1 = \{t_0, t_1, t_2\}$.*

We consider the following forget function $\tau : \lambda(t_i \in T_1) \cdot t$.

Then, any execution of P_0 corresponds to exactly one execution of P_1: for instance, $\langle(\ell_0, t), (\ell_1, t), (\ell_3, t), (\ell_4, t)\rangle$ corresponds to $\langle(\ell_0, t_0), (\ell_1, t_1), (\ell_2, t_1), (\ell_4, t_0)\rangle$. In particular, any transition step in P_0 is mimicked by a transition step in P_1 as mentioned in Definition 4.2.2, 2. Therefore, P_1 is a τ-partition of P_0.
Similarly, we can check that any execution, including one-step transitions of P_1 corresponds to some execution of P_1. Hence, P_1 is τ-complete with respect to P_0.
These two properties make P_1 a very useful extended system, in the analysis of P_0.
Intuitively, the extended system P_1 corresponds to a partition of P_0 obtained by delaying the merge in the exit of the conditional statement after the statement following the conditional, i.e. at point ℓ_4; this amounts to doing the following rewriting:

$$
\begin{array}{rl}
\ell_0: & \mathbf{if}(e)\{ \\
\ell_1: & \quad s_1 \\
 & \}\mathbf{else}\{ \\
\ell_2: & \quad s_2 \\
 & \} \\
\ell_3: & s_3 \\
\ell_4: & \ldots
\end{array}
\quad\longrightarrow\quad
\begin{array}{rl}
(\ell_0, t_0): & \mathbf{if}(e)\{ \\
(\ell_1, t_1): & \quad s_1; \\
(\ell_3, t_1): & \quad s_3 \\
 & \}\mathbf{else}\{ \\
(\ell_2, t_2): & \quad s_2; \\
(\ell_3, t_2): & \quad s_3 \\
 & \} \\
(\ell_4, t_0): & \ldots
\end{array}
$$

In particular, applying this partitioning to the example presented in Figure 4.1(a) would solve the imprecision. Indeed, it would allow proving that sgn cannot be equal to 0 *at ℓ_5, so that the division by sgn is safe; moreover, it allows proving that the absolute value of x computed in y is always positive.*
The System P_2 displayed in Figure 4.3(c) is also a complete partition of P_0. It amounts do performing a similar partitioning of the conditional structure without merging the traces at point ℓ_4. Such a partitioning would be more costly if applied to many **if***-statements in a large program.*
In fact, we can also note that P_2 is a complete partition of P_1.

Remark 4.2.1. Extending the notion of covering.

We may extend the definition of covering, by replacing the τ function with a relation $(\Rightarrow_\tau) \subseteq T \times T'$. Then, the function $\pi_\tau^{\mathbb{L}}$ becomes a relation $(\Rightarrow_\tau^{\mathbb{L}}) \subseteq \mathbb{L}_T \times \mathbb{L}_{T'}$.
Intuitively, $t \Rightarrow_\tau^{\mathbb{L}} t'$ means that the token t is "simulated" by t' in $P_{T'}$. Clearly, this definition is weaker, since a token t may be simulated by several tokens in $P_{T'}$.
The results in the following would extend to this weaker definition of covering system.

At this point, we do not require the set of partitions to be finite. This assumption is not required in order to prove the partitioning correct. However, we shall assume that $\mathbb{L}_T$ (hence, T) is finite whenever partitions must be computer representable; in particular, when defining the partitions used in the static analysis, T will always be supposed *finite.*

Trivial extension: We let $t_\epsilon \in \mathbb{T}$ and write $T_\epsilon = \{t_\epsilon\}$. The *trivial extension* of P is the extended system $P_\epsilon = (\mathbb{L}_\epsilon, \mathbb{S}^{\mathrm{i}}_\epsilon, \rightarrow_\epsilon)$, where:

- $\mathbb{L}_\epsilon = \mathbb{L} \times T_\epsilon$;
- $\mathbb{S}^{\mathrm{i}}_\epsilon = \{((l, t_\epsilon), \rho) \mid (l, \rho) \in \mathbb{S}^{\mathrm{i}}\}$;
- $((l_0, t_\epsilon), \rho) \rightarrow_\epsilon ((l_1, t_\epsilon), \rho) \iff (l_0, \rho) \rightarrow (l_1, \rho)$.

This extended system is isomorphic to P (the traces of both programs are equal up to isomorphism); it is the "simplest" extension of P. We write π^Σ_ϵ for the trivial mapping of traces of P_ϵ into traces of P.

4.2.2 Soundness of control partitioning

The ultimate goal of this chapter is to define an abstraction as the data of a partition (or covering) and an abstraction of the semantics of the corresponding extended system. Therefore, in the two following subsections, we set up an ordering, so as to compare the semantics of partitioned systems and build an ordering among partitioned systems.

The semantics of extended systems is defined in the usual way, as in Section 2.2.2. Furthermore, we propose to partition the semantics with the partitioned control states *including* the token (i.e., we choose $E = \mathbb{L}_T = \mathbb{L} \times T$), of the last state in the traces, which amounts to applying the same abstraction as $\alpha_{\mathbb{P}(\mathbb{L})}$ (Section 4.1.1) in the case of the extended system:

Definition 4.2.3. Partitioned semantics.

If P_T is the extended system $(T, \mathbb{S}^{\mathrm{i}}_T, \rightarrow_T)$, we let $[\![P_T]\!]^{\mathrm{p}}$ be the partitioned semantics defined by:

$$[\![P_T]\!]^{\mathrm{p}} = \alpha_{\mathfrak{P}(\delta_{\mathbb{L}_T})}([\![P_T]\!])$$

where $\delta_{\mathbb{L}_T}$ is defined by:

$$\begin{array}{llll} \delta_{\mathbb{L}_T}: & \mathcal{P}(\Sigma) & \rightarrow & (\mathbb{L}_T \rightarrow \mathcal{P}(\Sigma)) \\ & \mathcal{E} & \mapsto & \lambda((l, t) \in \mathbb{L}_T) \cdot \{\sigma \in \mathcal{E} \mid \exists \rho \in \mathbb{M},\ \sigma = \langle \ldots, ((l, t), \rho)\rangle\} \end{array}$$

The properties of covering (resp. partitioning, complete) systems extend to their semantics, as pointed out in the following lemma (the definitions for covering, partitioning and complete extended systems were designed so as to achieve these properties): for instance, a complete partition P_T of $P_{T'}$ provides a unique counterpart σ for any trace σ' of $P_{T'}$. In the following, we consider the programs $P_T = (T, \mathbb{S}^{\mathrm{i}}_T, \rightarrow_T)$ and $P_{T'} = (T', \mathbb{S}^{\mathrm{i}}_{T'}, \rightarrow_{T'})$, and $\tau : T \rightarrow T'$.

Lemma 4.2.2. Semantic adequation – traces.

Then:

- *If P_T is a τ-covering of $P_{T'}$, then:*

$$\forall \ell' \in \mathbb{L}_{T'},\ \forall \sigma' \in [\![P_{T'}]\!]^{\mathrm{P}}(\ell'),\ \exists \ell \in \mathbb{L}_T,\ \left\{ \begin{array}{l} \ell' = \pi_\tau^{\mathbb{L}}(\ell) \\ \exists \sigma \in [\![P_T]\!]^{\mathrm{P}}(\ell),\ \sigma' = \pi_\tau^{\Sigma}(\sigma) \end{array} \right.$$

- *If P_T is a τ-partition of $P_{T'}$, then:*

$$\forall \ell' \in \mathbb{L}_{T'},\ \forall \sigma' \in [\![P_{T'}]\!]^{\mathrm{P}}(\ell'),\ \exists!(\ell, \sigma) \in \mathbb{L}_T \times \Sigma_T,\ \left\{ \begin{array}{l} \ell' = \pi_\tau^{\mathbb{L}}(\ell) \\ \sigma \in [\![P_T]\!]^{\mathrm{P}}(\ell), \\ \sigma' = \pi_\tau^{\Sigma}(\sigma) \end{array} \right.$$

- *If P_T is τ-complete with respect to $P_{T'}$, then:*

$$\forall \ell \in \mathbb{L}_T,\ \forall \sigma \in [\![P_T]\!]^{\mathrm{P}}(\ell),\ \pi_\tau^{\Sigma}(\sigma) \in [\![P_{T'}]\!]^{\mathrm{P}}(\pi_\tau^{\mathbb{L}}(\ell))$$

Proof.

The proofs for these properties are similar, so we consider the last one only.
Therefore, we assume that P_T is τ-complete with respect to $P_{T'}$, and that $\ell \in \mathbb{L}_T, \sigma \in [\![P_T]\!]^{\mathrm{P}}(\ell)$, and we attempt to prove that $\pi_\tau^{\Sigma}(\sigma) \in [\![P_{T'}]\!]^{\mathrm{P}}(\pi_\tau^{\mathbb{L}}(\ell))$.
We write $\sigma = \langle s_0, \ldots, s_n \rangle$ and $\forall i,\ s'_i = \pi_\tau^{\mathbb{S}}(s_i)$ (so that $\sigma' = \langle s'_0, \ldots, s'_n \rangle = \pi_\tau^{\Sigma}(\sigma)$).

- First, we prove by induction on the length of σ that $\sigma' \in [\![P_{T'}]\!]$:
  - $s_0 \in \mathbb{S}_T^{\mathrm{i}}$; since P_T is τ-complete with respect to $P_{T'}$, $s'_0 = \pi_\tau^{\mathbb{S}}(s_0) \in \mathbb{S}_{T'}^{\mathrm{i}}$;
  - Let $i \in \mathbb{N},\ 0 \leq i < n$. Since $\sigma \in [\![P_T]\!]$, $s_i \rightarrow_T s_{i+1}$; hence, $s'_i \rightarrow_{T'} s'_{i+1}$, because P_T is τ-complete with respect to $P_{T'}$.
- Second, we prove that $\pi_\tau^{\Sigma}(\sigma) \in [\![P_{T'}]\!]^{\mathrm{P}}(\pi_\tau^{\mathbb{L}}(\ell))$: since $\sigma \in [\![P_T]\!]^{\mathrm{P}}(\ell)$, $\sigma \in [\![P_T]\!]$; hence, $\pi_\tau^{\Sigma}(\sigma) \in [\![P_{T'}]\!]$ (as proved in the first point). Moreover, σ' ends at point $\pi_\tau^{\mathbb{L}}(\ell)$, since $s'_n = \pi_\tau^{\mathbb{S}}(s_n)$. Hence, $\pi_\tau^{\Sigma}(\sigma) = \sigma' \in [\![P_{T'}]\!]^{\mathrm{P}}(\pi_\tau^{\mathbb{L}}(\ell))$

The cases of partitioning and covering systems are similar. □

Let Γ_τ be the function defined as:

$$\begin{array}{llll} \Gamma_\tau : & (\mathbb{L}_T \rightarrow \mathcal{P}(\Sigma_T)) & \rightarrow & (\mathbb{L}_{T'} \rightarrow \mathcal{P}(\Sigma_{T'})) \\ & \Phi & \mapsto & \lambda(\ell' \in \mathbb{L}_{T'}) \cdot \bigcup\{\pi_\tau^{\Sigma}(\Phi(\ell)) \mid \ell \in \mathbb{L}_T,\ \tau(\ell) = \ell'\} \end{array}$$

Here are a few trivial properties of the Γ_τ functions:

Lemma 4.2.3. Properties of Γ_τ.

For all τ, Γ_τ is monotone.
If $\tau_0 : T_0 \rightarrow T_1,\ \tau_1 : T_1 \rightarrow T_2$, then $\Gamma_{\tau_1 \circ \tau_0} = \Gamma_{\tau_1} \circ \Gamma_{\tau_0}$.

The following theorem comes as a straightforward consequence of Lemma 4.2.2; it is an important step in proving the soundness of the partitioning abstractions.

Theorem 4.2.4. Semantic adequation.

With the above notations:

- *If P_T is a τ-partition or a τ-covering of $P_{T'}$, then* $[\![P_{T'}]\!]^{\mathrm{P}} \subseteq \Gamma_\tau([\![P_T]\!]^{\mathrm{P}})$ (soundness).
- *If P_T is τ-complete with respect to $P_{T'}$, then* $\Gamma_\tau([\![P_T]\!]^{\mathrm{P}}) \subseteq [\![P_{T'}]\!]^{\mathrm{P}}$ (completeness).
- *Hence, if P_T is a τ-complete partition of $P_{T'}$, or a τ-complete covering of $P_{T'}$, then* $[\![P_{T'}]\!]^{\mathrm{P}} = \Gamma_\tau([\![P_T]\!]^{\mathrm{P}})$ (adequation).
- *If P_T is a partitioning system of $P_{T'}$, then:*

$$\forall \ell, \ell' \in \mathbb{L}_T,\ \ell \neq \ell' \Longrightarrow \Gamma_\tau([\![P_T]\!]^{\mathrm{P}})(\ell) \cap \Gamma_\tau([\![P_T]\!]^{\mathrm{P}})(\ell) = \emptyset$$

4.2.3 Pre-ordering properties of partitions

In the following, we use an ordering among partitions. Therefore, we study the pre-ordering properties of the following relations, among extended transition systems:

- “is a covering of” (for some forget function τ);
- “is a partition of” (for some forget function τ);
- “is complete with respect to” (for some forget function τ).

Then, we can prove that, any such ordering $\preccurlyeq$ is transitive:

Lemma 4.2.5. Transitivity.

Let us consider $P_T = (T, \mathbb{S}^{\mathrm{i}}_T, \rightarrow_T)$, $P_{T'} = (T', \mathbb{S}^{\mathrm{i}}_{T'}, \rightarrow_{T'})$, and $P_{T''} = (T'', \mathbb{S}^{\mathrm{i}}_{T''}, \rightarrow_{T''})$. Furthermore, we consider the forget functions $\tau : T \rightarrow T'$, and $\tau' : T' \rightarrow T''$. Then:

- *if P_T is a τ-covering (resp. τ-partition) of $P_{T'}$ and $P_{T'}$ is a τ'-covering (resp. τ-partition) of $P_{T''}$, then P_T is a $(\tau' \circ \tau)$-covering (resp. $(\tau' \circ \tau)$-partition) of $P_{T''}$.*
- *if P_T is τ-complete with respect to $P_{T'}$ and $P_{T'}$ is τ'-complete with respect to $P_{T''}$, then P_T is $(\tau' \circ \tau)$-complete with respect to $P_{T''}$.*

Proof.

We can first remark that $\pi^{\mathbb{L}}_{\tau' \circ \tau} = \pi^{\mathbb{L}}_{\tau'} \circ \pi^{\mathbb{L}}_{\tau}$ (and similarly for the other forget functions). Let us prove the second point (transitivity of completeness).

- Let $s \in \mathbb{S}^{\mathrm{i}}_T$. Then, $\pi^{\mathbb{S}}_\tau(s) \in \mathbb{S}^{\mathrm{i}}_{T'}$, since P_T is τ-complete with respect to $P_{T'}$. Moreover, $\pi^{\mathbb{S}}_{\tau' \circ \tau}(s) = \pi^{\mathbb{S}}_{\tau'} \circ \pi^{\mathbb{S}}_\tau(s) \in \mathbb{L}^{\mathrm{i}}_{T''}$, since $P_{T'}$ is τ'-complete with respect to $P_{T''}$.
- Let $s_0, s_1 \in \mathbb{S}_T$, such that $s_0 \rightarrow_T s_1$. Again, we apply successively the two assumptions of completeness and derive $\pi^{\mathbb{S}}_\tau(s_0) \rightarrow_{T'} \pi^{\mathbb{S}}_\tau(s_1)$ (since P_T is τ-complete with respect to $P_{T'}$ and then $\pi^{\mathbb{S}}_{\tau' \circ \tau}(s_0) \rightarrow_{T''} \pi^{\mathbb{S}}_{\tau' \circ \tau}(s_1)$, since $P_{T'}$ is τ'-complete with respect to $P_{T''}$.

The proof of the first point is similar. □

Moreover, the relations mentioned above are clearly reflexive

Lemma 4.2.6. Reflexivity.

Let $P_T = (T, \mathbb{S}^{\mathrm{i}}_T, \rightarrow_T)$ and $\tau : T \rightarrow T$; $t \mapsto t$. Then, clearly P_T is a τ-covering (resp. partition) of P_T and P_T is τ-complete with respect to itself.

Such an ordering should allow to compare the *precision* of partitions (yet, note that the more precise partition is the greater element, instead of the smaller, as is usually the case in static analysis) and to define valid *computational orderings* [CC92b], which we will illustrate in the next section.

4.3 Trace Partitioning Abstract Domains

The last section described the partitioning of transition systems. We now build on top of this material a partitioning domain of traces, with partitions based on the control flow, and define further partitioning abstractions, by composing stores and numerical abstractions.

4.3.1 The trace partitioning domain

Definition of the basis: In this section, we assume that a transition system $P = (\mathbb{L}, \mathbb{S}^{\mathrm{i}}, \rightarrow)$ is given, and we consider the complete coverings of P; we write $\mathfrak{B}$ for the set of the extended systems which satisfy these properties.

First, we let $\preccurlyeq$ be the order among extended systems defined by:

$$P_{T_0} \preccurlyeq P_{T_1} \iff \exists \tau : T_1 \rightarrow T_0,\ P_{T_1} \text{ is a } \tau\text{-covering}$$

As remarked in Section 4.2.3, we may choose other definitions for $\preccurlyeq$, such as:

$$P_{T_1} \preccurlyeq P_{T_1} \iff \exists \tau : T_1 \rightarrow T_0, \begin{cases} P_{T_1} \text{ is a } \tau\text{-partition of } P_{T_0} \\ P_{T_1} \text{ is } \tau\text{-complete with respect to } P_{T_0} \end{cases}$$

In case the property on the right side is satisfied, we also write $P_{T_0} \preccurlyeq_\tau P_{T_1}$ for τ, so as to make τ explicit.

The trivial extension of P is clearly the least element of $\mathfrak{B}$ for $\preccurlyeq$.

Note that other choices for $\mathfrak{B}$ and $\preccurlyeq$ could have been made and would have allowed to prove the same results in the following.

Definition 4.3.1. The ordering over the basis.
We showed in Example 4.2.1 that the systems P_0, P_1 and P_2 are such that:

$$P_0 \preccurlyeq P_1 \preccurlyeq P_2$$

The domain: At this point we can define the trace partitioning domain. An element of this domain should denote:

- a covering P_T of the original transition system;
- and a semantic denotation for each control state ℓ of the covering P_T:
  - in the basic domain, this denotation shall be a set of traces ending at point ℓ);
  - in the abstract domain, this denotation shall be an invariant in $D^\sharp_{\mathbb{M}}$.

More formally:

Definition 4.3.1. Trace partitioning domain.
An element of the trace partitioning domain is a tuple (T, P_T, Φ), where:

- *$T \in \mathfrak{T}$;*
- *P_T denotes a complete covering $(T, \mathbb{S}^{\mathrm{i}}_T, \rightarrow_T)$ of P;*
- *Φ is a function $\Phi : \mathbb{L}_T \rightarrow \mathcal{P}(\Sigma_T)$.*

We write $\mathbb{D}$ for the set of such tuples.
Let $(T_0, P_{T_0}, \Phi_0), (T_1, P_{T_1}, \Phi_1) \in \mathbb{D}$. Then, we write $(T_0, P_{T_0}, \Phi_0) \leqslant_\tau (T_1, P_{T_1}, \Phi_1)$ –or, for short $(T_0, P_{T_0}, \Phi_0) \leqslant (T_1, P_{T_1}, \Phi_1)$– if and only if:

- *$P_{T_0} \preccurlyeq_\tau P_{T_1}$ for τ;*
- *$\Phi_0 \subseteq \Gamma_\tau(\Phi_1)$.*

It follows from the results presented in Section 4.2.3 that $\leqslant$ defines a pre-ordering on $\mathbb{D}$.

The concretization function: The concretization of an element (T, P_T, Φ) of $\mathbb{D}$ is a set of traces of the initial system, which is computed by:

1. merging all the partitions together, by projecting Φ onto the trivial extension P_ϵ of P (i.e., applying function Γ_{τ_ϵ}) and then collapsing the partitions with $\gamma_{\mathfrak{P}(\mathbb{L}_T)}$;
2. applying the isomorphism π^Σ_ϵ between traces of P_ϵ and P.

It is defined formally in the following definition:

Definition 4.3.2. Concretization function.

We let $\gamma_{\mathbb{P}}$ be the concretization function defined by

$$\gamma_{\mathbb{P}} = \pi^\Sigma_\epsilon \circ \gamma_{\mathfrak{P}(\mathbb{L}_T)} \circ \Gamma_{\tau_\epsilon}$$

Or equivalently, by:

$$\begin{array}{rlcl} \gamma_{\mathbb{P}} : & \mathbb{D} & \rightarrow & \Sigma \\ & (T, P_T, \Phi) & \mapsto & \{\pi^\Sigma_\epsilon(\sigma) \mid \exists \ell \in \mathbb{L}_T,\ \sigma \in \Phi(\ell)\} \end{array}$$

Clearly, this function is monotone.

Soundness of the partitioned systems: A last, trivial yet very important remark is that the partitioning of the initial system is sound:

Theorem 4.3.1. Soundness of control partitioning.

Let $(T_0, P_{T_0}), (T_1, P_{T_1}) \in \mathfrak{T} \times \mathfrak{B}$, such that $P_{T_0} \preccurlyeq_\tau P_{T_1}$. Then, $(T_0, P_{T_0}, [\![P_{T_0}]\!]^{\mathrm{P}}) \leqslant_\tau (T_1, P_{T_1}, [\![P_{T_1}]\!]^{\mathrm{P}})$.
In particular, in case $(T_0, P_{T_0}) = (T_\epsilon, P_\epsilon)$, then we get the soundness with respect to the original *transition system: $[\![P]\!] \subseteq \gamma_{\mathbb{P}}(T_1, P_{T_1}, [\![P_{T_1}]\!]^{\mathrm{P}})$.*

Proof.
The first point follows from Theorem 4.2.4; the second is a corrolary of the first point. □

This domain structure can be related to the cofibered domain structure defined in [Ven96]. More precisely, the element of the basis fixes a partition of the original system,

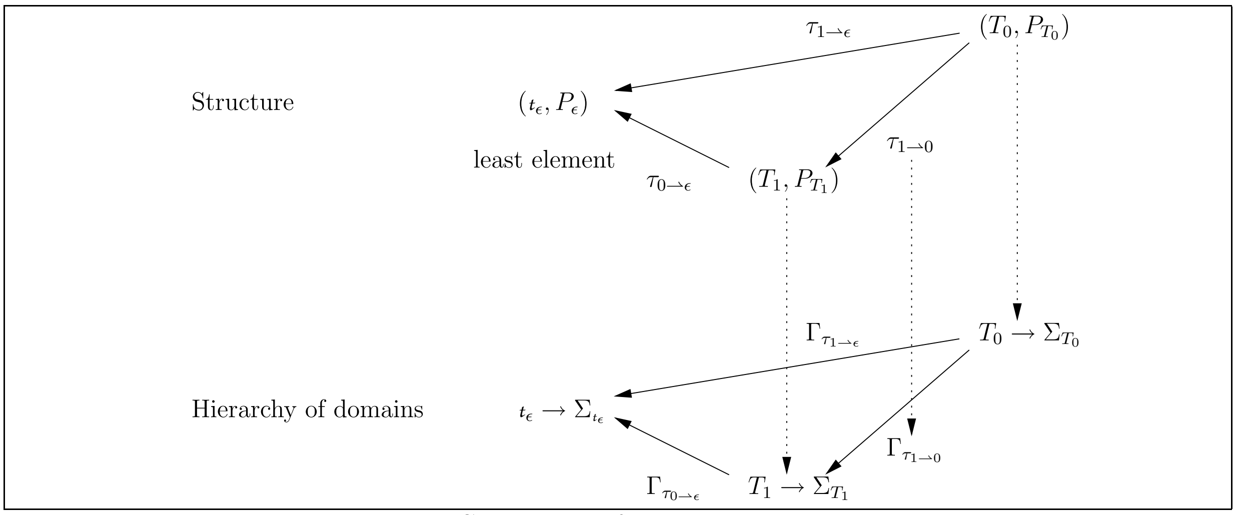


Figure 4.4: Structure of the partitioning domain

and the last argument of the tuple corresponding to an element of the domain $\mathbb{D}$ provides a semantic denotation defined in a domain relative to the basis element. Figure 4.4 gives an overall intuition about the structure of the partitioning domain $\mathbb{D}$.

The presentation in [Ven96] relies on categories; we use orderings instead, but the principle is similar: the structure of the basis provides the frame for a hierarchy of domains. The comparison of elements across different domains can be done thanks to the projection functions Γ_τ provided by the ordering on the basis.

Gain in precision: Let $(T, P_T, \Phi) \in \mathbb{D}$ be an element of the domain. This element describes the *same* set of traces as the initial program P. However, it allows for a more precise description of sets of traces ending at each control state than the usual abstractions (i.e., the $\alpha_{\mathbb{P}(\mathbb{L})}$ abstraction defined in Section 4.1.1), if there exists a control state $l \in \mathbb{L}$, $\sigma, \sigma' \in [\![P]\!]$, such that σ and σ' both end at l but are not in the same partitions, when mapped into the extended system P_T. This gain in precision really pays off, when a further abstraction (such as the abstraction defined by $\gamma_{\mathbb{M}}$) is composed, as done in the next subsection.

Comparison with other approaches to partitioned systems: Our approach considerably generalizes the trace partitioning technique of [HT98], since we leave the choice of partitions as a *parameter*: various partitioning strategies can be implemented (for instance, we allow the merge of partitions).

The path sensitive techniques [HR80] proposed in data flow analysis context do not allow for *abstractions of sets of paths* to be considered. In our settings, a token stands for an approximation for a set of paths, which renders the design of analyses more flexible.

Other authors proposed to perform a partitioning of memory states or to convert part of the data into control structures, as can be done for booleans [JHR99]. However, this solution presents several drawbacks in our opinion. In particular, the relations partitions are based on may not be found straightforwardly in the memory states; in the other hand, a partitioning guided by the conditions is rather intuitive. Another drawback comes from the fact that the method exposed in [JHR99] is based on a refinement process, which would not be so effective in the case of the ASTRÉE analyzer. By contrast this approach seem to be more effective for the analysis of synchronous programs.

The following subsections express fundamental properties of $\mathbb{D}$:

- composition of further abstractions (such as the abstraction of sets of stores into collections of predicates), in Section 4.3.2;
- application to static analysis and definition of widening operators on such domains, in Section 4.3.3;
- implementation of efficient analyzers in Section 4.3.4.

4.3.2 Composing store abstraction

We derive from Definition 4.3.1 the definition of a new partitioning abstraction, by abstracting sets of stores into collections of constraints in the same way as in Section 3.1.1. Therefore, we assume that an abstraction $(D^\sharp_{\mathbb{M}}, \sqsubseteq)$ is defined for representing sets of stores is defined, together with a concretization function $\gamma_{\mathbb{M}} : D^\sharp_{\mathbb{M}} \to \mathcal{P}(\mathbb{M})$, which defines the meaning of a set of abstract constraint as the set of stores which satisfy them.

The partitioning abstract domain is derived from $\mathbb{D}$ by replacing functions mapping extended labels into sets of traces with functions mapping extended labels into elements of $D^\sharp_{\mathbb{M}}$:

Definition 4.3.3. Partitioning abstract domain.

An element of the partitioning abstract domain is a tuple $(T, P_T, \Phi^\sharp)$, where:

- *$T \in \mathfrak{T}$;*
- *P_T is a complete covering of P $(T, \mathbb{S}^{\mathrm{i}}_T, \to_T)$;*
- *$\Phi^\sharp$ is a function $\Phi^\sharp : \mathbb{L}_T \to D^\sharp_{\mathbb{M}}$.*

We write $\mathbb{D}^\sharp$ for the set of such tuples.

Remark 4.3.1. Representation of abstract values.

An abstract value is a value in $\mathbb{L}_T \to D^\sharp_{\mathbb{M}} = (\mathbb{L} \times T) \to D^\sharp_{\mathbb{M}}$. By curryfication; it is isomorphic to a value in $\mathbb{L} \to (T \to D^\sharp_{\mathbb{M}})$. This latter representation turns out to be very natural in practice: each control state corresponds to an abstract value in the partitioning domain $D^\sharp_{\mathbb{P},\mathbb{M}} = T \to D^\sharp_{\mathbb{M}}$, mapping partitioning tokens into sets of stores; hence, it allows to describe precisely the partitions associated to each program point.

The ordering is also inherited from Definition 4.3.1. Indeed, we let:

$$\begin{array}{rlcl} \Gamma^\sharp_\tau : & (\mathbb{L}_T \to D^\sharp_{\mathbb{M}}) & \to & (\mathbb{L}_{T'} \to D^\sharp_{\mathbb{M}}) \\ & \Phi^\sharp & \mapsto & \lambda(\ell' \in \mathbb{L}_{T'}) \cdot \bigsqcup\{\Phi(\ell) \mid \ell \in \mathbb{L}_T,\ \tau(\ell) = \ell'\} \end{array}$$

If the join operator $\sqcup$ of $D^\sharp_{\mathbb{M}}$ is not associative, commutative, the definition of $\Gamma^\sharp_\tau$ would not be unique, which would cause various technical complications; therefore, we assume that $\sqcup$ is associative and commutative in our presentation. Then:

Definition 4.3.4. Ordering.

Let $(T_0, P_{T_0}, \Phi^\sharp_0), (T_1, P_{T_1}, \Phi^\sharp_1) \in \mathbb{D}$, and a function $\tau : T_1 \to T_0$. Then, we write $(T_0, P_{T_0}, \Phi^\sharp_0) \leqslant^\sharp_\tau (T_1, P_{T_1}, \Phi^\sharp_1)$ (or, for short $(T_0, P_{T_0}, \Phi^\sharp_0) \leqslant^\sharp (T_1, P_{T_1}, \Phi^\sharp_1)$) if and only if:

- *$P_{T_0} \preccurlyeq_\tau P_{T_1}$ for τ;*
- *$\Phi^\sharp_0 \sqsubseteq \Gamma^\sharp_\tau(\Phi^\sharp_1)$.*

It follows from the results presented in Section 4.2.3 that $\leqslant$ defines a pre-ordering on $\mathbb{D}$.

The concretization of an element of $\mathbb{D}^\sharp$ into an element of $\mathbb{D}$ applies the concretization function $\gamma_{\mathbb{M}}$ pointwise, i.e. by applying it to $\Phi^\sharp$.

Definition 4.3.5. Concretization.

$$\begin{array}{rlcl} \gamma^\sharp_{\mathbb{P}} : & \mathbb{D}^\sharp & \to & \mathbb{D} \\ & (T, P_T, \Phi^\sharp) & \mapsto & (T, P_T, \lambda(\ell \in \mathbb{L}_T) \cdot \gamma_{\mathbb{M}} \circ \Phi^\sharp(\ell)) \end{array}$$

We remark, that $(T, P_T, \Phi^\sharp)$ may provide a better approximation of $[\![P]\!]$ than an element in $D^\sharp = \mathbb{L} \to D^\sharp_{\mathbb{M}}$ whenever the extended systems distinguishes traces of P, i.e., if there exists a control state ℓ, and $\sigma, \sigma' \in [\![P]\!]$ such that σ and σ' both end at ℓ and are in different partitions, when mapped into traces of P_T.

In the other hand, any approximation for $[\![P]\!]$ in $D^\sharp$ can be translated in an *equivalent* abstraction in $(T, P_T, \Phi^\sharp)$, for *any* choice of (T, P_T). As a consequence, we expect the partitioning domain to provide results at least as good as the non partitioning domain, and strictly better results when the (T, P_T) allows to distinguish real traces of P.

At this point, we can state a few remarks, which should give a better understanding of the structure of the partitioning domain.

Remark 4.3.2. Computational ordering and precision ordering.

The ordering introduced in Definition 4.3.4 is essentially a computational ordering [CC92b]. Indeed, an analysis starts with a coarse partition, defined by the program control structure and then may perform some refinements of the system. When a refinement is performed, the basis element is replaced with a greater element, and so is the current abstract invariant. Therefore, the abstract computation should produce monotone sequences of elements for the ordering of Definition 4.3.4.

Next subsection proposes the definition of an extrapolation operator based on the same computational order.

Remark 4.3.3. Direction of the ordering on the basis.

We pointed out in that the ordering among elements of the basis is an inverse *for the precision ordering the end of section 4.2.3: the greater for $\preccurlyeq$, the more precise the partition. Therefore, one may suggest using the opposite ordering. However, this approach has several drawbacks:*

- *It would not capture the precision ordering better than the current ordering. Indeed, we may have $(T_0, P_{T_0}, \Phi_0^\sharp) \leqslant^\sharp (T_1, P_{T_1}, \Phi_1^\sharp)$ even though P_{T_0} and P_{T_1} are not comparable for $\preccurlyeq$; opposing the ordering on the basis would not help here.*
- *It would be possible to write the analysis so that it starts with a* completely partitioned system *(which may not be easy to define, depending on the instantiation of the partitioning framework) and use the opposite ordering as a computational ordering also (the analysis should merge partitions so as to ensure termination): however, we found this idea less intuitive; in particular, it is easier to reason about* creating *partitions instead of not deleting partitions.*

4.3.3 Static analysis with partitioning and a widening operator

The domain introduced in Section 4.3.2 allows to carry out a static analysis of P, with a partitioning domain. However, several approaches to such analyses are feasible:

- **static partitioning** relies on the choice of a fixed partition;
- **dynamic partitioning** allows for the partition to be changed during the static analysis.

The latter approach is more powerful but may also result in a more involved implementation. In particular, in case infinitely many partitions might be chosen and different partitions can be used for successive iterations in an abstract fixpoint computation, the termination of the analysis shall be enforced by the use of a *widening* operator. For instance, it may start analyzing a loop by unrolling the first iterates and decide to give up the unrolling at some point, so as to guarantee termination of the analysis.

The definition of a widening operator on $\mathbb{D}^\sharp$ is necessary when infinite or very large sets of partitions shall be used, and when (quick) termination is required, e.g. for static analysis. This issue would not occur in case the set of partitions was chosen once for all.

We propose to define a widening operator for $\mathbb{D}^\sharp$ by:

- choosing a widening $\nabla_{\mathbb{M}}$ over $D_{\mathbb{M}}^\sharp$;
- choosing a widening $\nabla_{\mathfrak{B}}$ over the basis;
- defining a pairwise widening over $\mathbb{D}^\sharp$.

Formally, the widening operator for the partitioning domain is defined by:

Definition 4.3.6. Widening for the partitioning domain.

If $(T_0, P_{T_0}, \Phi_0^\sharp), (T_1, P_{T_1}, \Phi_1^\sharp) \in \mathbb{D}$, then, we let:

$$(T_0, P_{T_0}, \Phi_0^\sharp)\nabla_{\mathrm{p}}(T_1, P_{T_1}, \Phi_1^\sharp) = (T_2, P_{T_2}, \Phi_2^\sharp)$$

where:

- $P_{T_2} = P_{T_0} \nabla_{\mathfrak{B}} P_{T_1}$, *so that* $P_{T_0} \preccurlyeq_{\tau_0} P_{T_2}$ *and* $P_{T_1} \preccurlyeq_{\tau_1} P_{T_2}$*;*
- $\Phi_2^\sharp = (\Phi_0^\sharp \circ \tau_0) \nabla_{\mathbb{M}} (\Phi_1^\sharp \circ \tau_1)$ *(pointwise application of* $\nabla_{\mathbb{M}}$ *to elements of* $\mathbb{L}_{T_2} \longrightarrow D_{\mathbb{M}}^\sharp$*).*

Indeed, this approach leads to a widening over the partitioning abstract domain, as shown in the following theorem:

Theorem 4.3.2. Widening for partitioning domains.
The operator ∇_{p} *is a widening operator on* $\mathbb{D}$*, in the sense of Definition 2.3.2.*

Proof.
Proving point 1 in Definition 2.3.2 is straightforward, so we consider point 2.
Let $(T_n, P_{T_n}, \Phi_n^\sharp)_{n \in \mathbb{N}}$ be a sequence elements of $\mathbb{D}$, and $(T'_n, P_{T'_n}, \Phi'^\sharp_n)_{n \in \mathbb{N}}$ be defined as:

$$\begin{array}{rcl} (T'_0, P_{T'_0}, \Phi'^\sharp_0) & = & (T_0, P_{T_0}, \Phi^\sharp_0) \\ (T'_{n+1}, P_{T'_{n+1}}, \Phi'^\sharp_{n+1}) & = & (T'_n, P_{T'_n}, \Phi'^\sharp_n) \nabla_{\mathrm{p}} (T_n, P_{T_n}, \Phi^\sharp_n) \end{array}$$

Then:

- by definition of the widening over the basis $\nabla_{\mathfrak{B}}$, the element of the basis stabilizes after a finite number of iterations: $\exists n \in \mathbb{N},\ \forall m \in \mathbb{N},\ m \geq n \Longrightarrow P_{T_m} = P_{T_n}$.
- if we consider the subsequence $(T'_m, P_{T'_m}, \Phi'^\sharp_m)_{m \in \mathbb{N}, m \geq n}$, then $\forall m \geq n,\ T'_m = T'_n \wedge P_{T'_m} = P_{T'_n}$ and the sequence of the last arguments form a widening sequence in $\mathbb{L}_{T'_n} \longrightarrow D^\sharp_{\mathbb{M}}$; $\mathbb{L}_{T'_n}$ is finite and $\nabla_{\mathbb{M}}$ is a widening over $D^\sharp_{\mathbb{M}}$, therefore this sequence is ultimately stationary.

This proves that the sequence $(T'_n, P_{T'_n}, \Phi'^\sharp_n)_{n \in \mathbb{N}}$ is ultimately stationary; hence, ∇_{p} is a widening operator over $\mathbb{D}$. $\square$

Again, the proof of the widening operator can be compared with the definition of a widening on cofibered domains [Ven96]. Basically, a widening operator for $\mathbb{D}$ should stabilize the basis first (i.e., enforce the termination of the refinement of the partition), and then stabilize the image in the abstract domain $D^\sharp_{\mathbb{M}}$; therefore, an alternate definition for ∇_{p} would delay the widening in $D^\sharp_{\mathbb{M}}$ until the element of the basis reaches a limit.

4.3.4 Denotational style partitioning static analysis

The design of static analyzers as abstractions of the denotational semantics of statements was proposed in Section 3.2.5. In particular, we showed that this design allows for natural and efficient iteration strategies. Therefore, we propose to adapt this scheme to partitioning analyses.

Partitioning denotational semantics: First, we apply the "from point to point" denotational abstraction $\alpha_{\iota\mathcal{F}[\ell_\vdash, \ell_\dashv]}$.

More precisely, we consider in this subsection an extended system P_T, such that $P \preccurlyeq_\tau P_T$, and let $\ell_\vdash, \ell_\dashv \in \mathbb{L}$. The concrete denotational semantics from $\ell_\vdash$ to $\ell_\dashv$ maps an "input" state at $\ell_{\vdash}$ to the set of possible "output" states at $\ell_{\dashv}$. Hence, the denotational semantics in the extended system should map tuples made of a partitioning token and a store into similar tuples:

Definition 4.3.7. Partitioned denotational semantics.

We define the abstraction function $\alpha_{t\mathcal{F}\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]} : \mathcal{P}(\Sigma) \rightarrow ((\mathbb{T} \times \mathbb{M}) \rightarrow \mathcal{P}(\mathbb{T} \times \mathbb{M}))$*, where* $\alpha_{t\mathcal{F}\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]}(\mathcal{E})$ *is defined by:*

$$\begin{array}{rrcl} \alpha_{t\mathcal{F}\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]}(\mathcal{E}) : & (\mathbb{T} \times \mathbb{M}) & \rightarrow & \mathcal{P}(\mathbb{T} \times \mathbb{M}) \\ & (t_{\vdash}, \rho_{\vdash}) & \mapsto & \{(t_{\dashv}, \rho_{\dashv}) \mid \exists \sigma \in \mathcal{E},\ \sigma = \langle (((\ell_{\vdash}, t_{\vdash}), \rho_{\vdash}), \ldots, ((\ell_{\dashv}, t_{\dashv}), \rho_{\dashv}) \rangle \} \end{array}$$

We write $\gamma_{t\mathcal{F}\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]}$ *for the corresponding concretization function.*
Last, the partitioned denotational semantics *is* $\alpha_{t\mathcal{F}\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]}(\llbracket P_T \rrbracket^{\mathrm{P}})$.

Static, abstract partitioning denotational semantics: The denotational-style static analyzer of Section 3.2.5 was derived as an abstraction of the denotational semantics; therefore, we propose to derive a static analyzer for the partitioned system in the same way. However, we should note a slight difference: in Definition 4.3.7, an initial state consists in a pair made of a partitioning token and a store. Hence, the abstract semantics follows the same scheme:

Definition 4.3.8. Partitioned abstract denotational semantics.

We write $D^{\sharp}_{\mathbb{P},\mathbb{M}}$ *for* $T \rightarrow D^{\sharp}_{\mathbb{M}}$. *A function* $\llbracket P_T \rrbracket^{\sharp}_{\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]} : D^{\sharp}_{\mathbb{P},\mathbb{M}} \rightarrow D^{\sharp}_{\mathbb{P},\mathbb{M}}$ *is a* sound abstract semantics *of* P_T*, between* $\ell_{\vdash}$ *and* $\ell_{\dashv}$ *if and only if:*

$$\begin{array}{l} \forall (t, \rho), (t', \rho') \in \mathbb{T} \times \mathbb{M},\ \forall d_p \in D^{\sharp}_{\mathbb{P},\mathbb{M}} \\ \left. \begin{array}{l} (t', \rho') \in \alpha_{t\mathcal{F}\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]}(\llbracket P_T \rrbracket)(t, \rho) \\ \rho \in d_p(t) \end{array} \right\} \Longrightarrow \rho' \in \llbracket P_T \rrbracket^{\sharp}_{\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]}(d_p)(t') \end{array}$$

In this sense, $\llbracket P_T \rrbracket^{\sharp}_{\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]}$ should be an approximation of the denotational semantics introduced in Definition 4.3.7.

The partitioned denotational abstract semantics is sound with respect to the standard semantics of the initial system:

Theorem 4.3.3. Soundness of the static partitioning analysis.

Let $(T, P_T) \in \mathfrak{B}$ *such that* $(T_\epsilon, P_\epsilon) \preccurlyeq_{\tau} (T, P_T)$.
Let $d_t \in D^{\sharp}_{\mathbb{P},\mathbb{M}}$, $(t, \rho) \in \mathbb{T} \times \mathbb{M}$ *such that* $\rho \in d_t(t)$. *Moreover, we let* $\rho' \in \alpha_{t\mathcal{F}[\ell_{\vdash},\ell_{\dashv}]}(\llbracket P \rrbracket)(\rho)$.
Then, there exists t' *such that:*

$$\rho' \in \llbracket P_T \rrbracket^{\sharp}_{\mathbb{P}[\ell_{\vdash},\ell_{\dashv}]}(d_p)(t')$$

Proof.
The above result follows from the soundness of the control partitioning (Theorem 4.3.1) and the soundness of the abstract semantics $[\![P_T]\!]^\sharp_{\mathbb{P}[\ell_\vdash,\ell_\dashv]}$ (Definition 4.3.8). $\square$

In practice, an abstract semantics $[\![P_T]\!]^\sharp_{\mathbb{P}[\ell_\vdash,\ell_\dashv]}$ is defined in a similar way as the abstract semantics of statements described in Section 3.2.5, and in Figure 3.3.

Moreover, we can remark that the abstract semantics $[\![P_T]\!]^\sharp_{\mathbb{P}[\ell_\vdash,\ell_\dashv]}$ may postpone the computation of abstract joins so as to approximate flows in distinct partitions. This ability allows in many cases for a greater precision (even if a local improvement in precision does not always guarantee a global improvement, since several abstract operators including widening usually are not monotone).

Definition 4.3.2. Denotational style abstraction of a if-statement.
We consider the program introduced in Example 4.2.1. In particular, this program is equivalent to the transition system P_0, displayed in Figure 4.3(a). We consider the partition defined by the system P_1 (Figure 4.3(b)): the analysis partitions the traces depending on the branch of the **if**-*statement they visited until point ℓ_4 (the partitions are merged at this point).*
We present the static analysis of various statements in this piece of code (the analysis is carried out on P_1):

- *statement s_1 (true branch of the conditional): the only partitions before and after this statement is t_1, to $[\![s_1]\!]^\sharp_{\mathbb{P}[\ell_1,\ell_3]}$ is a function:*

$$[\![s_1]\!]^\sharp_{\mathbb{P}[\ell_1,\ell_3]} : (\{t_1\} \to D^\sharp_{\mathbb{M}}) \longrightarrow (\{t_1\} \to D^\sharp_{\mathbb{M}})$$

  (the analysis propagates the partition t_1);
- *conditional structure (statement $s = \mathbf{if}(e)\, s_1 \,\mathbf{else}\, s_2$): it splits the partition t_0 into two sets of traces corresponding to t_1 and t_2; hence, $[\![s]\!]^\sharp_{\mathbb{P}[\ell_1,\ell_3]}$ is a function:*

$$[\![s]\!]^\sharp_{\mathbb{P}[\ell_1,\ell_3]} : (\{t_0\} \to D^\sharp_{\mathbb{M}}) \longrightarrow (\{t_1, t_2\} \to D^\sharp_{\mathbb{M}})$$

- *statement s_3 (statement right after the conditional): it inputs two partitions corresponding to t_1 and t_2 and outputs similar partitions; however, the partitions are merged right after the analysis of the statement (at point ℓ_4), so we can write down $[\![s_3]\!]^\sharp_{\mathbb{P}[\ell_3,\ell_4]}$ as a function:*

$$[\![s_3]\!]^\sharp_{\mathbb{P}[\ell_3,\ell_4]} : (\{t_1, t_2\} \to D^\sharp_{\mathbb{M}}) \longrightarrow (\{t_0\} \to D^\sharp_{\mathbb{M}})$$

- *the whole program inputs and outputs only one partition, corresponding to t_0, so its abstract semantics is a function:*

$$[\![P_1]\!]^\sharp_{\mathbb{P}[\ell_1,\ell_3]} : (\{t_0\} \to D^\sharp_{\mathbb{M}}) \longrightarrow (\{t_0\} \to D^\sharp_{\mathbb{M}})$$

Making the partitioning dynamic: The above definition introduces a static form of partitioning: the analysis of the statement may not change the partitions, e.g. by refining the system. Therefore, we propose a new definition for an abstract semantics for statements, which may refine the partitions.

First, we define a new partitioning abstract domain for approximating sets of stores *and* partitions:

Definition 4.3.9. Domain for dynamic partitioning.

An element of the domain is a tuple (T, P_T, d_T), where:

- $T \in \mathbb{T}$;
- P_T *is a complete covering* $(T, \mathbb{S}^{\mathrm{i}}_T, \to_T)$ *of the initial system* P;
- $d_p \in D^\sharp_{\mathbb{P},\mathbb{M}}$ *is such that* $\forall \iota \in \mathbb{T} \setminus T,\ d_p(\iota) = \bot$.

We write $D^\sharp_{\delta\mathbb{P},\mathbb{M}}$ for this domain; the ordering is the pointwise extension of the orderings on the basis and on $D^\sharp_{\mathbb{M}}$.

The latter condition ensures that d_p assigns invariants to "relevant" tokens only: the invariant corresponding to a token not in T (i.e., not in the current extended system) should be $\bot$.

A partitioning abstract semantics can be defined as follows:

Definition 4.3.10. Dynamic partitioning analysis.

The abstract semantics *of P_T between $l_\vdash$ and $l_\dashv$ is a function $[\![P_T]\!]^\sharp_{\mathbb{P}[l_\vdash, l_\dashv]} : D^\sharp_{\delta\mathbb{P},\mathbb{M}} \to D^\sharp_{\delta\mathbb{P},\mathbb{M}}$ such that, if $(T, P_T, d_T), (T', P_{T'}, d_{T'}) \in D^\sharp_{\delta\mathbb{P},\mathbb{M}}$ are such that $(T', P_{T'}, d'_{T'}) = [\![P_T]\!]^\sharp_{\mathbb{P}[l_\vdash, l_\dashv]}(T, P_T, d_T)$, then, there exists $\tau : T' \to T$ satisfying the following conditions:*

- $P_{T'}$ *refines* P_T, *i.e.* $P_T \preccurlyeq_\tau P_{T'}$;
- $d'_{T'}$ *approximates the output of $P_{T'}$ at $l_\dashv$ when the input at $l_\vdash$ is described by d_T in the previous system in a sound manner, which is expressed by the following condition, where $d_{T'} = d_T \circ \tau$:*

$$\begin{array}{l}\forall (\iota, \rho), (\iota', \rho') \in \mathbb{T} \times \mathbb{M}, \\ \left.\begin{array}{l} (\iota', \rho') \in \alpha_{\iota\mathcal{T}\mathbb{P}[l_\vdash, l_\dashv]}([\![P_{T'}]\!])(\iota, \rho) \\ \rho \in d_{T'}(\iota) \end{array}\right\} \Longrightarrow \rho' \in d'_{T'}(\iota')\end{array}$$

Note that the soundness of the "abstract transfer function" in the second of point of Definition 4.3.10 is expressed in the refined system: the input invariant d_T is refined into $d_{T'}$ first, and then the abstract transition is performed in T'.

This abstract semantics is sound as well:

Theorem 4.3.4. Soundness of the dynamic partitioning analysis.

Let $(T, P_T) \in \mathfrak{B}$ such that $(T_\epsilon, P_\epsilon) \preccurlyeq_\tau (T, P_T)$. Let $d_t \in D^\sharp_{\mathbb{P},\mathbb{M}}$, $(\iota, \rho) \in \mathbb{T} \times \mathbb{M}$ such that $\rho \in d_t(\iota)$. We write $(T', P_{T'}, d'_{T'})$ for the result of the analysis $[\![P_T]\!]^\sharp_{\mathbb{P}[l_\vdash, l_\dashv]}(T, P_T, d_T)$. Moreover, we let $\rho' \in \alpha_{\iota\mathcal{F}[l_\vdash, l_\dashv]}([\![P]\!])(\rho)$. Then, there exists ι' such that

$$\rho' \in d'_{T'}(\iota')$$

Proof.

Similar to the proof of 4.3.3. □

Again, the core of the soundness of the analysis lies in the definition of the abstract transformer in the refined transition system, which should soundly approximate the partitioning of the transitions of the original system.

Definition 4.3.3. Denotational style abstraction of a if-statement.

Example 4.3.2 demonstrates the analysis of a conditional statement, based on a static partitioning of P_0 into P_1.

In the case of dynamic partitioning, the main difference is that, before the analysis of the conditional, the system under consideration is P_0 and that the analysis refines P_0 into P_1 at point l_1 (beginning of the conditional). After this refinement, the book-keeping of the partitions is the same as in Example 4.3.2.

Chapter 5

Control-based partitioning

In this chapter, we describe the implementation of a domain for control flow-based trace partitioning inside the ASTRÉE analyzer, and to provide experimental evidence of the efficiency of the approach. This domain is enabled in all analyses, so as to improve the precision of ASTRÉE. We provide a few examples as well, so as to show how partitioning contributes to improving the partition.

We give a quick description of the ASTRÉE analyzer in Section 5.1. Section 5.2 describes the analysis, by instantiating the framework introduced in Chapter 4 and providing abstract transfer functions for the partitioning and merging of traces. Section 5.3 provides facts about the implementation (in particular, about the strategies used in order to determine when to perform partitioning); it concludes with experimental results and a comparison with related work.

5.1 The ASTRÉE analyzer

ASTRÉE is an academic static analyzer developed in the École Normale Supérieure and in the École Polytechnique by Bruno BLANCHET, Patrick COUSOT, Radhia COUSOT, Jérôme FERET, Laurent MAUBORGNE, Antoine MINÉ, David MONNIAUX and myself. The ASTRÉE static analyzer aims at proving the absence of runtime errors in large, embedded programs, written in C [ANS99]. Various aspects of the ASTRÉE static analyzer were described in [BCC+02, BCC+03a, CCF+05]. A user manual was written as well [BCC+03b].

5.1.1 The programs analyzed by ASTRÉE

The development of the ASTRÉE analyzer started in fall 2001. First positive results were reported in early 2002, with the analysis with **0 false alarms** of some 10 000 LOCs example program.

At this point, the analyzer was designed in order to analyze large embedded applications, written in C. The main specificities of these programs are:

- the **large size:** up to more than 100 000 LOCs, and 10 000 global variables;
- the **control structure:** these programs implement synchronous applications translated into C programs. Form more information about synchronous programming, we refer the reader to the definition of the synchronous languages Lustre [HCRP91], Esterel [BG92], and Signal [ABG95]. More precisely, they consist in a large loop, which should be executed every t milliseconds. For each iteration of the main loop, some routines read a large set of inputs, perform computations involving both the inputs and some state variables storing the state of the system and send some outputs:

```
while(true){
    ≀Loop executed every $t$ ms∫

    ≀Read inputs from sensors∫
    input($x_{\mathrm{in}}$); input($y_{\mathrm{in}}$); ...

    ≀Computation of the new internal state∫
    $X_0 = \ldots ; X_1 = \ldots ;$

    ≀Sending of outputs∫
    $x_{\mathrm{out}} = \ldots ; y_{\mathrm{out}} = \ldots ;$
}
```

- the **floating point computations:** most of the routines involve floating point computation, including linear filtering, non linear control with feed-back, interpolations from input values, limiters, conversions into/from integer values and bit fields...
- the **large number of conditions, and control flow stored in boolean variables**: a large number of boolean variables store the state of the computer and greatly impact the control flow in the core of the main loop (e.g., initialization and reset variables, raising edge detectors...).

These specificities led to crucial implementation choices, so as to ensure scalability first; and then, to refine the analysis whenever a false alarm was discovered. This strategy allowed us to discover the nature of the predicates required for inferring precise invariants of these families of programs and then, to implement the adequate abstract domains. Whenever the addition of a new domain was required, we strove to maintain the scalability of the analysis.

This approach allowed us to report on the successful analysis of a large part of the Airbus A340 aircraft fly-by-wire device in 2003. Several versions of the next generation of fly-by-wire systems (developed for the Airbus A380 aircraft) were successfully analyzed in 2004 and 2005. A more detailed report of the performances of ASTRÉE will be provided in Section 5.3.3.

At the time of the writing of this thesis, other families of programs are being considered as well.

5.1.2 The purpose of the analysis

The purpose of the analysis is to discover *all* possible runtime errors (i.e., failed operation causing the computer to crash or to switch to an abnormal state), detailed below. Of course, ASTRÉE over-approximates the behaviors of the program being analyzed; therefore, it may report *false alarms*, i.e. ASTRÉE may report not being able to prove the correctness of some critical operation, despite no real execution crashes at this point. In case ASTRÉE raises no alarm after analyzing a program, then the program can be considered safe in the sense that it should neither crash nor produce an erroneous value at runtime. Of course, this conclusion depends on the soundness of the analyzer and on the correctness of the assumptions made about the system (including, compliance to the C semantics [ANS99], to the choices made compiler, correctness of the assumptions made about the input values such as their range...).

The ultimate goal in the design of the analyzer is to reduce the number of false alarms, while preserving the efficiency of the analysis. In case the analysis generates some alarms, the program may be erroneous or the alarms may be due to the approximation inherent in the static analysis. Therefore, all alarms should be investigated. Part III deals with the issue of the investigation of the alarms.

The purpose of ASTRÉE is to discover any possible runtime error, where the definition of "runtime error" collects the following cases:

- **fatal errors and unspecified behaviors** in the sense of the ANSI'99 C semantics [ANS99], including memory errors (e.g., array index out of bounds), integer division by 0. Some architecture dependent behaviors may not be considered errors (then, the analyzer should comply with the specification of the target architecture, and of the compiler).
- **generation of infinite floating point values** (floating point overflows) or of the "Not-A-Number" floating-point value (e.g., after a division 0/0).
- **non-compliance with programming guidelines**, which forbid, e.g. the overflow of `short` integer variables out of the rangle $[-32\,768, 32\,767]$, even though the result may be well-defined on some specific platform: for instance, if short integers values are stored in 32 registers during computations, then a value resulting from an overflow may be exactly representable, so the behavior of the system may not be affected.
- **failure of a user-defined assertion**.

It follows from the success of the analysis of a program that the only possible interrupts are clock ticks, which is an essential requirement for the safety of synchronous programs.

5.1.3 The analyzer

Overall structure of the analyzer: A run of the ASTRÉE analyzer consists in a sequence of phases:

1. **Preprocessing and preparation of the analysis:**

- **parsing**, and **merging** of programs implemented in multiple files: this phase produces a very low level syntax tree, without explicit types;
- **typing** and synthesis of a higher level syntax tree, for a limited, yet rather large subset of ANSI C 99 [ANS99] (for instance, some peculiar C-initializers are rejected at this stage);
- **code simplification**, including Kildall constant propagation [Kil73] and removal of dead statements;
- verification of the **semantic definition** of the code **independently from the evaluation order** (since the [ANS99] norm leaves the evaluation order implementation undefined): more precisely, we check that the order side effects are performed in should not depend on the C compiler; this property is crucial for the analysis to be sound (the analyzer cannot simulate many different execution orders);
- **inclusion of analysis directives**, which should guide the analysis of the code, either by suggesting hints to the relational abstract domains about what packs of variables relations should be computed for (aka, “packing strategy”), and partitioning directives (we discuss the insertion of partitioning directives in Section 5.3.2;
- translation into a last **internal representation**, with all expressions “flattened”: at this point, there should be no control flow in expressions (i.e., lazy logical operators, conditional evaluations) containing side-effects; moreover, function calls are expanded in atomic operations;

2. **Analysis and output of the results:**
   - **initialization** of the abstract domains, following the parameterization of the analyzer;
   - **analysis**, i.e. computation of invariants for the program, following Section 3.2.5;
   - **checking of the safety of the critical operations**, in the check phase of the iteration;
   - optional **export to invariant** into files.

A large number of options allow to tune the analysis (by enabling or disabling abstract domains, modifying iteration strategy parameters), to configure the parallel mode or disable it, the output of the analyzer (verbosity of the text messages, enabling or disabling of warnings for some alarms), the export of invariants (by selecting what domains and what control states information should be exported for), the pre-processing steps and to require the analyzer to produce various debug outputs.

A separate tool provides an interface for visualizing invariants (it requires the invariants being exported into a file in the end of the analysis).

The iterator: The iterator is designed in the denotational style presented in Section 3.2.5. However, the analysis of a statement outputs not only an invariant but also some reports for the alarms (in case the analyzer does not prove the execution of the statement is safe) during the last iteration of the analysis (check mode), and it also allows to store local invariants to the disk (producing a result similar to those of the interpreter presented in Section 3.1.2).

Hence, the interpreter function carries out several parameters, which impact greatly the iteration strategy and the application of the transfer functions:

- a flag indicates the iteration mode, i.e. whether or not the analyzer should check the safety of critical operations and report alarms (in the case of a loop, the analyzer should not do so before it computes an over-approximation of the semantics of the loop; hence, only the last iteration is performed in "check" mode);
- some flags describing the state of the iterator; in the case of the analysis of loops, they guide the widening strategy and the definition of transfer functions (e.g., reductions).

The abstract domains: ASTRÉE is based on a large collection of abstract domains. The core of the abstract domain aims at approximating sets of stores in a similar way as $D^\sharp_{\mathbb{M}}$ in Section 3.1.1. In fact, this domain is split into two parts: a *structure domain* describes a mapping of concrete program variables into abstract memory cells, and a *relational domain*, which approximates sets of functions from abstract memory cells into values. This abstraction is performed after the following abstractions:

1. partitioning abstraction of traces, as explained in this Chapter;
2. abstraction of forward branching flows, following a principle based on continuation semantics: an abstract element encloses not only the current abstract flow, but also abstract branching flows, together with label they are branching to (function exit, control state after a **cases**-statement...).

The numerical abstract domain is built as a reduced product of a series of domains; each of them allows to express specific kinds of constraints:

- the **interval domain** [CC77] collects range constraints of the form:

  $$x \in [a, b]$$

  All safety properties of interest (except user defined assertions) can be expressed with such invariants; however, this domain does not allow for *precise* invariants to be inferred.
- the **octagon domain** [Min01, Min04b] expresses relations of the form

  $$\pm x \pm y \leq c$$

  This domain allows for relational invariants to be computed for pieces of code implementing limiters, absolute value...
- a dedicate domain performs the translation of arithmetic expressions into **interval linear forms** [Min04a], i.e. expressions of the form $\sum_k I_k \cdot x_k$, where I_k is an interval and x_k a variable for all k. This domain has several purposes:
  - allow the transfer functions of relational domains like octagons to be used, even in the case of complex expressions;

  - take rounding errors into account in the abstract interpretation of floating point expressions.
- a **symbolic domain** [Min04b] collects symbolic equalities relations among variables, which can be used so as to perform a *reduction* of the abstract values of other domains;
- a **domain for the analysis of filters** [Fer04b] represents predicates useful for proving the stability of filters. For instance, in the case of a second order filter, the value of x_n of x at iteration n is computed from the previous values using a formula like $x_n = a \star x_{n-1} + b \star x_{n-2}$. If the filter is stable, we would usually be able to prove that any pair made of two successive values lies in an ellipsoïd. However, when this proof need to be performed automatically, it may require the use of polyhedra with a large number of faces (very costly abstraction, complex transfer functions); therefore, a specific domain represents ellipsoïd predicates explicitly and detects filters.
- a domain of **arithmetic geometric progressions** [Fer04a] allows to bound slowly diverging floating point computations, so as to prove that they do not diverge after a long (yet not infinite) time of execution.
- a domain of **boolean relations** using the principles of BDDs [Bry86] in order to express relations among boolean variables and mixed relations (relations among boolean and arithmetic variables). In the latter case, the elements of the domain consist in trees with boolean relations at the nodes and numerical relations at the leaves.

Implementation: Most of the analyzer is written in Objective Caml [OCa]; however, it uses a few libraries written in C. At the time of the writing of this thesis, it amounts to 70 000 lines of Objective Caml and 9 000 lines of C code (mainly, the octagon library, and some low level routines used for setting the rounding mode).

It is noticeable that the soundness of the analysis does not depend on the architecture the analysis is performed on: at this time, Astrée has been successfully used on a large number of architectures, including Intel Pentium, AMD 64, Sun UltraSparc, and PowerPC architectures, running various operating systems including Linux, Unix, Microsoft Windows and Mac OSX.

We provide detailed results about performances in execution time, memory usage, and precision in Section 5.3.3.

5.2 Partitioning Analysis

We now introduce the trace partitioning domain integrated in the Astrée analyzer, together with some examples showing how it contributes to improving precision.

5.2.1 Partitioning criteria

First, we list the criteria for trace partitioning in ASTRÉE :

1. **Partitioning of conditional structures,** by delaying the merge of flows in the end of the conditional;
2. **Partitioning of loop structures,** by distinguishing the first iterations in the analysis of the loop body and delaying the merge of flows after the end of the loop. This criteria allow for:
   - more precise invariants to be derived in the first iterations, thanks to unrolling;
   - relations between numbers of iterations and values to be inferred and used after the loop, thanks to the delayed abstract join;
3. **Partitioning guided by the value of a variable** x at some point ℓ (the partitions are computed at point ℓ and not modified by an assignment to x): this partitioning is similar to a case analysis based on the value of a variable (this partitioning scheme is most useful when dealing with weak updates, and array accesses);
4. **Inlining of functions** (as suggested in Section 4.1.1);
5. **Merge of partitions:** the cost of successive creations of partitions would be prohibitive in practice. For instance, the partitioning of a conditional structure multiplies by 2 the number of partitions in the current flow, so a series of n conditional structures would lead to a 2^n blow-up, which is not acceptable (no scalable analysis can afford an exponential cost). Therefore, we avail ourselves the possibility of merging together unnecessary partitions (i.e. partitions which are not expected to lead to further improvements in precision), in any order.

Some of these cases could be handled by rewriting the code. This approach is depicted in Figure 5.1(a), in the case of the partitioning of a conditional structure (case 1), as suggested in Example 4.2.1: the statements following the conditional are duplicated in the end of both branches. Case 2 (loops) and case 4 (function inlining) could be handled in a similar manner. For instance, Figure 5.1(b) displays the rewriting equivalent to the unrolling of the first iteration of a loop.

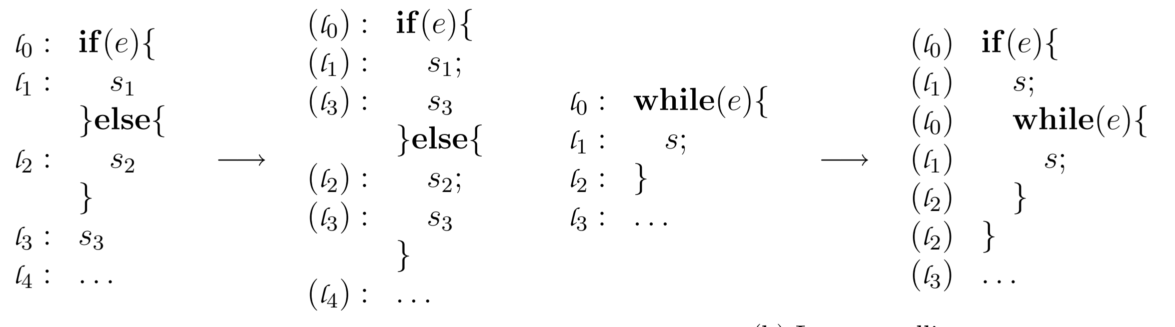
However, we show in Section 5.2.3 that the design of a trace partitioning domain was preferable, so that finer partitions can be handled.

5.2.2 Application of trace partitioning

Before we set up the partitioning domain, we provide a few examples, so as to show how the main criteria for partitioning introduced in Section 5.2.1 are useful, in ASTRÉE.

Linear interpolation function, via indirection arrays: We consider the case of the interpolation function $\mathrm{f}_{\mathrm{lin}}$ described in Figure 5.2 first.

The piece of code for this function determines what formula should be used by localizing in what range x can be found, using a loop and an array of input values. Then, two arrays contain the coefficients which should be used in order to compute the value of



```
ℓ0 :  if(e){                 (ℓ0) :  if(e){
ℓ1 :      s1                 (ℓ1) :      s1;
      }else{                 (ℓ3) :      s3
ℓ2 :      s2        ⟶                }else{
      }                      (ℓ2) :      s2;
ℓ3 :  s3                     (ℓ3) :      s3
ℓ4 :  ...                            }
                             (ℓ4) :  ...
```

(a) Partitioning of a conditional

```
ℓ0 :  while(e){              (ℓ0)  if(e){
ℓ1 :      s;                 (ℓ1)      s;
ℓ2 :  }             ⟶        (ℓ0)      while(e){
ℓ3 :  ...                    (ℓ1)          s;
                             (ℓ2)      }
                             (ℓ2)  }
                             (ℓ3)  ...
```

(b) Loop unrolling

Control states in parentheses denote partitioned control states.

Figure 5.1: Code rewriting

$\mathrm{f}_{\mathrm{lin}}(x)$. Clearly, the output of this function is bounded: $\forall x,\ \mathrm{f}_{\mathrm{lin}}(x) \in [-1, 2]$.

However, inferring this most precise range is not feasible with a standard interval analysis, even if we partition the traces depending on the values of i at point ℓ_3. Let us try with $-100 \leq x \leq 0$: then, we get $i \in \{0, 1\}$ at point ℓ_3. The range for y at point ℓ_4 is $[-0.5 + 0.5 \times (-100.), -0.5] \equiv [-50.5, -0.5]$ (this range is obtained in the case $i = 1$; the case $i = 0$ yields $y = -1$). Accumulating such huge imprecision during the analysis may cause the properties of interest (e.g. the absence of runtime errors or the range of output values) not to be proved. We clearly see that some relations between the value of x and the value of i are required here.

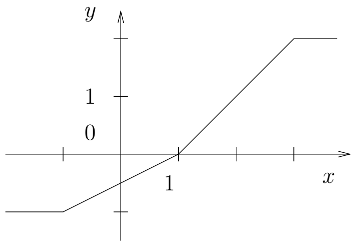
Our approach is to partition the traces according to the number of iterations in the loop. Indeed, if the loop is not iterated, then $i = 0$ at point ℓ_3 and $x < -1$; if it is iterated exactly once, then $i = 1$ at point ℓ_3 and $-1 \leq x \leq 1$ and so forth. This approach yields the most precise range. Let us resume the analysis, with the initial constraint $-100 \leq x \leq 0$. The loop is iterated at most once and the partitions at point ℓ_3 give:

- 0 iteration: $i = 0$; $x < -1$; $y = -1$
- 1 iteration: $i = 1$; $-1 \leq x \leq 0$; $-1 \leq y \leq -0.5$.

Therefore, the resulting range is $y \in [-1, -0.5]$, which is the optimal range (i.e. exactly the range of all output values that can be observed in concrete executions).

This optimal result is obtained thanks to a partitioning of the traces by the number of iterations in the loop. The partitions can be merged after the output of the function, since they should not result in any further gain in precision.

Linear interpolation function, via discretization: The second example consists in another kind of interpolation function: the input value is disctretized, and then a formula depending on the discretized value is applied to it. More precisely, if $\lfloor x \rfloor = n$, and f is the function to approximate, then the interpolation $\mathrm{f}_{\mathrm{lin}}$ returns $\mathrm{f}(n) + (x - n) \times (\mathrm{f}(n + 1) - \mathrm{f}(n))$. From the mathematical point of view, it is a particular case of the



$$y = \begin{cases} -1 & \text{if } x \leq -1 \\ -0.5 + 0.5 \times x & \text{if } -1 \leq x \leq 1 \\ -1 + x & \text{if } 1 \leq x \leq 3 \\ 2 & \text{if } 3 \leq x \end{cases}$$

(a) Function

```
ℓ0 :  int i = 0;
ℓ1 :  while(i < n && x > tx[i + 1])        tc = {0; 0.5; 1; 0}
ℓ2 :         i ++ ;                         tx = {0; −1; 1; 3}
ℓ3 :  y = tc[i] × (x − tx[i]) + ty[i]       ty = {−1; −0.5; −1; 2}
ℓ4 :  . . .
```

(b) Implementation

Figure 5.2: Linear interpolation, via indirection arrays

interpolation function considered in the previous paragraph, where the values of the array tx are successive integer values. In the example presented in Figure 5.3, the array ty is such that $ty[n] = \mathrm{f}(n)$. Any interpolation based on a regular partition of a bounded range could be implemented in a similar way, by applying a linear function to the argument so as to recover a partition of the form $0, 1, \ldots, n$.

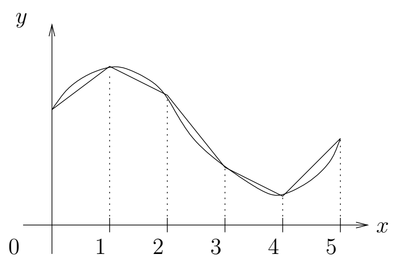
We found that this kind of interpolations were rather common, e.g. for approximating trigonometric functions. For the same reason as in the case of the previous interpolation function, the computation of a precise range for the output of $\mathrm{f}_{\mathrm{lin}}$ requires some precise relation between n and x.

However, the possible values for n cannot be related to distinct control flow paths; therefore, we propose to perform a partitioning guided by the *value of n computed at ℓ_1*. Doing the same partitioning at point ℓ_2 would not allow for relations between x and i to be obtained.

5.2.3 The domain

Need for a trace partitioning domain: As we pointed out in Section 5.2.1, some of the partitioning configurations could have been carried out by rewriting the code. However, we enumerate a number of reasons in favor of the design of a real domain.

First, the **"syntactic transformation" approach is limiting**. In particular, it would not allow to represent and handle large sets of partitions in the same way as a dedicate domain would:



$$\begin{array}{l} \text{if } n \leq x < n+1, \text{ where } n \in \mathbb{Z} \\ y \;=\; \mathrm{f}(n) + (x-n) \times (\mathrm{f}(n+1) - \mathrm{f}(n)) \end{array}$$

(a) Mathematical definition

$$\begin{array}{ll} \ell_0: & \mathrm{int}\, n = 0; \\ \ell_1: & n = \mathbf{cast}_{\mathrm{float}\to\mathrm{int}} x; \\ \ell_2: & y = ty[n] + (x - \mathbf{cast}_{\mathrm{int}\to\mathrm{float}} n) * \\ & \quad (ty[n+1] - ty[n]) \\ \ell_3: & \ldots \end{array}$$

(b) Implementation

Figure 5.3: Linear interpolation function, via discretization

- a domain allows to represent *more* partitions than mere syntactic rewriting, since not all possible partitions need to be generated during the analysis despite the syntactic approach would require to generate them all prior to the analysis;
- a syntactic rewriting of the code would be inherently *static*, which is not practically compatible with very large sets of partitions. For instance, a partitioning guided by the values of a variable may generate a huge number of partitions if the variable may take a large number of values (e.g., thousands of values); in this case, a built-in strategy would not perform the partitioning (by not sending the partitioning order to the domain), whereas the decision whether to partition or not would need to be made prior to the analysis in the case of syntactic partitioning. In this case, the implementation of a partitioning domain allows to tune the partitioning strategy during the analysis, so that better decisions can be taken about whether or not some partitions should be generated.

Secondly, as we pointed out above, **the partitions** sometimes **need to be merged** together. Currently, where and which partitions are merged is the result of some strategies (Section 5.3.2). However, the last partition created may not be merged first, which implies that the structure of partitions should be found in abstract elements (as a consequence, the code rewriting approach would fail to offer the same flexibility).

Thirdly, in some cases, partitions could be created **in a lazy way only** not only for cost reasons, as in the following cases:

- in the case of a function call, where the function is the result of the dereference of a pointer, the control flow can only be known at analysis time;

- some strategies may determine that a loop should be unrolled n times and the analysis may prove that after $m < n$ iterations the execution of the loop terminates; then a syntactic unrolling would not make sense.

Last, the **inspection of analysis results** is easier, when the invariants can be related to the original program, with accurate partition names (i.e., tokens in the scheme of Chapter 4). Rewriting large pieces of code as suggested in Figure 5.1 would make the understanding of the result of static analyses more difficult, since the user would have to relate the invariants computed for the transformed program to the original program. By contrast, the values of the partitioning domain should tell what partitions numerical constraints correspond to, thanks to the partitioning tokens.

Elements: We now define formally the instantiation of the framework presented in Chapter 4 corresponding to the criteria listed in Section 5.2.1.

Intuitively, the creation of a partition corresponds to a partitioning directive, as defined in Section 5.2.1. We provide the formal definition of directives in Figure 5.4(a). The name of each directive corresponds very intuitively to a criterion listed in Section 5.2.1, except for the last one: the directive $\mathfrak{part}\langle\mathbf{None}\rangle$ is included here for the sake of implementation only, and stands for a void directive (we explain the use of this directive in Section 5.3.1).

The name of a partition (i.e., token corresponding to it, in the sense of Section 4.2.1) consists in the series of the partitioning directives encountered before creating this partition. We give the formal definition for tokens in Figure 5.4(b). We note that each partitioning directive encloses a control state, which stands for the point the partition was created at. The directive $\mathfrak{part}\langle\mathbf{None}\rangle$ stands for a void directive, and as such, it can be removed from tokens without changing their meaning: in other words, the equality on tokens is defined modulo removal of void directives (i.e., $\mathfrak{part}\langle\mathbf{None}\rangle :: \mathfrak{part}\langle\mathbf{If}, \ell, b\rangle = \mathfrak{part}\langle\mathbf{If}, \ell, b\rangle$).

For instance, in the case of a conditional at point ℓ, two partitions are created right after the testing of the condition, corresponding to the directives "true branch of the conditional at point ℓ" and "false branch of the conditional at point ℓ". When these partitions are merged, these directives are removed from the names of the partitions.

As usual, we write $D^\sharp_{\mathbb{M}}$ for the domain for representing sets of stores (Section 3.1.1). In the same way as in Section 4.3.4, the domain $D^\sharp_{\mathbb{P},\mathbb{M}}$ is defined as $\mathbb{T} \rightarrow D^\sharp_{\mathbb{M}}$.

Hints (or directives) in the code: A pre-processing phase inserts directives as special commands in the source code. We do not introduce them formally here (the directives are represented as text between braces in programs). Intuitively, directives in the code cause directives to be added in tokens (partition creation) or be deleted from tokens (partition merge).

Widening: The set of tokens is clearly infinite, since the length of tokens as sequences of directives is not bounded. Even in case we limit the length of tokens the number of tokens is very large: indeed, if we fix $\ell \in \mathbb{L}$ and $x \in \mathbb{X}$, the number of directives of the

d	$::=$	$\mathfrak{part}\langle \textbf{If}, \ell, b\rangle$	traces in the b branch of the conditional at point ℓ
	$\mid$	$\mathfrak{part}\langle \textbf{While}, \ell, n\rangle$	traces with exactly n iterations in the loop at point ℓ
	$\mid$	$\mathfrak{part}\langle \textbf{While}, \ell, > n\rangle$	traces with more than n iterations in the loop at point ℓ
	$\mid$	$\mathfrak{part}\langle \textbf{Val}, \ell, x = n\rangle$	traces such that $x = n$ at point ℓ
	$\mid$	$\mathfrak{part}\langle \textbf{Fun}, \ell, f\rangle$	traces calling f at point ℓ
	$\mid$	$\mathfrak{part}\langle \textbf{None}\rangle$	void directive

(a) Directives (notation for directives: $d \in \mathcal{D}$)

ι	$::=$	ϵ	empty stack, initial partition
	$\mid$	$d :: \iota'$	addition of a directive on top of ι'

(b) Tokens ($\iota \in \mathbb{T}$)

Figure 5.4: Naming partitions

form $\mathfrak{part}\langle \textbf{Val}, \ell, x = n\rangle$ is equal to the number of integer values in the language (i.e., in practice 2^{32}). Therefore, the termination of the analysis should rely on a widening operator, designed as in Section 4.3.3.

In practice,

- the widening operator on the basis forbids the synthesis of arbitrary long tokens, by preventing the generation of tokens containing two directives corresponding to the same control point: basically, this operator interrupts the generation of partitions;
- the generation of partitions after a directive recommending the partitioning guided by the values of a variable x is performed only if the size of the set of possible values for x determined by the analysis is small enough (e.g., below 1000);
- the current partitioning strategy is designed so as not to keep partitions beyond the scope they should improve the precision in; this strategy allows to merge partitions soon enough, so that the widening operator does not need to collapse partitions down (widening is applied at loop heads only [Bou93]).

5.2.4 Structure of the abstract interpreter

As stated in Section 5.1.3, the iterator consists in a function mapping statements into abstractions of their denotational semantics, as defined in Section 3.2.5. As a consequence, the design of the abstract interpreter follows the principle described in Section 4.3.4: the abstract interpretation $[\![s]\!]^\sharp$ of a statement s should map a pair $(P_T, d_T) \in \mathfrak{B} \times D^\sharp_{\mathbb{P},\mathbb{M}}$, where $\forall \iota \notin T,\ d_T(\iota) = \bot$ into a pair $(P_{T'}, d'_{T'}) \in \mathfrak{B} \times D^\sharp_{\mathbb{P},\mathbb{M}}$, where $P_{T'}$ is a refinement of P_T and $d'_{T'}$ is an over-approximation of the output of s when applied to the input d_T (Definition 4.3.10).

The iterator of ASTRÉE does not keep track of the whole refined program P_T. Instead, it keeps track of the *current partitions*, i.e. of the tokens corresponding to a set of partitions covering the ongoing flows:

Definition 5.2.1. Ongoing token set.

The ongoing token set *corresponding to the abstract flow* $d_T \in D^\sharp_{\mathbb{P},\mathbb{M}}$ *is* $\mathbf{tokens}_T\langle d_T \rangle = \{t \in \mathbb{T} \mid d_T \neq \bot\}$.

This notion was implicitly illustrated in Example 4.3.2 (we described the partitioning abstract interpretation of an **if**-statement).

If (T, P_T, d_T) is the result of the static analysis of a statement, then, the property $\mathbf{tokens}_T\langle d_T \rangle \subseteq T$ is straightforward.

The abstract interpretation $[\![s]\!]^\sharp$ of a statement s simply maps an element $d_T \in D^\sharp_{\mathbb{P},\mathbb{M}}$ into a second element $d'_{T'} \in D^\sharp_{\mathbb{P},\mathbb{M}}$: all the information about the partitioning carried out by the analysis are enclosed in the d_T element.

This is a common advantage of denotational style abstract interpreters: this iteration scheme keeps only the information which are useful for the end of the analysis and discards the values which were useful only in the past and will not be required anymore. For instance, we remarked that the analyzer presented in Section 3.2.5 does not need to store invariants at every control point. The restriction to the set of tokens corresponding to the ongoing flows is similar.

This approach is feasible, since the partitioning tokens contain all the information about the transitions associated to them.

Last, we note that the pre-processing phase inserts hints in the code and selects this way a family of extended systems which may be used during the analysis. As a consequence, most of the partitioning decisions are made statically; the only decisions taken at analysis time are whether or not to obey to some directives. In this sense, the partitioning implemented in ASTRÉE is dynamic, but mostly determined statically; reducing the number of choices made at analysis time simplifies the implementation.

5.2.5 Transfer functions

We consider three kinds of transfer functions:

- the "partition creation" transfer function generate new partitions;
- the "partition merge" folds partitions together;
- the "standard" transfer functions (i.e., which are not specific to partitioning analyses) stand for e.g., abstract assignments, condition testing...

"Usual" transfer function, e.g. assignment: we extend pointwisely the usual transfer functions presented in Section 3.1.1 to $D^\sharp_{\mathbb{P},\mathbb{M}}$.

Partition creation: we let $\mathit{generate} : \mathcal{D} \times D^{\sharp}_{\mathbb{P},\mathbb{M}} \rightarrow D^{\sharp}_{\mathbb{P},\mathbb{M}}$ be the partition creation abstract transfer function. It inputs a directive ∂ and an abstract element $d \in D^{\sharp}_{\mathbb{P},\mathbb{M}}$ and adds the directive ∂ to all ongoing tokens in d. Formally, it outputs an element d', defined by:

$$\begin{cases} \mathbf{tokens}_T\langle d' \rangle = \{(\partial :: \iota) \mid \iota \in \mathbf{tokens}_T\langle d \rangle\} \\ \forall \iota \in \mathbf{tokens}_T\langle d \rangle,\ d'(\partial :: \iota) = d(\iota) \end{cases}$$

Partition merge: we let $\mathit{merge} : \mathcal{P}(\mathcal{D}) \times D^{\sharp}_{\mathbb{P},\mathbb{M}} \rightarrow D^{\sharp}_{\mathbb{P},\mathbb{M}}$ be the transfer function for merging partitions. It folds partitions by removing any directive in $\mathcal{D}$ for the partition names (tokens). Therefore merge inputs a set of directives D and an abstract element d and returns a new abstract element d', where any reference to the directives in D are removed. Formally, if $D = \{\partial\}$, then d' is defined by:

$$\begin{cases} (\partial_{i_0} :: \ldots :: \partial_{i_m}) \in \mathbf{tokens}_T\langle d' \rangle \iff \begin{cases} (\partial_0 :: \ldots :: \partial_n) \in \mathbf{tokens}_T\langle d \rangle \\ \{i_k \mid k \in [\![0, m]\!]\} = \{i \in [\![0, n]\!] \mid \partial_i \neq \partial\} \\ i_0 < \ldots < i_m \end{cases} \\ \text{With the above notations, } d'(\partial_{i_0} :: \ldots :: \partial_{i_m}) = d(\partial_0 :: \ldots :: \partial_n) \end{cases}$$

The above definition extends straightforwardly to the general case (D not necessarily a singleton).

Definition 5.2.1. Transfer functions in a partitioning analysis.

Figure 5.5 displays a simple piece of code, containing an **if***-statement (Figure 5.5(a)). The pre-processing phase of* ASTRÉE *includes some directives in the code, which specify what partitions should be created. We assume that the strategies recommend to partition the traces in the beginning of the* **if***-statement and to merge the partitions at point l_5, as shown in Figure 5.5(b)).*

- *at point l_0, only one partition exist; it corresponds to the void token ϵ;*
- *when entering the* **if***-statement, the analyzer creates two partitions corresponding to the directives $\mathfrak{part}\langle \mathbf{If}, l_0, \mathbf{true}\rangle$ (true branch) and $\mathfrak{part}\langle \mathbf{If}, l_0, \mathbf{false}\rangle$ (false branch): at this step it applies the transfer functions $d \mapsto \mathit{generate}(\mathfrak{part}\langle \mathbf{If}, l_0, \mathbf{true}\rangle, d)$ and $d \mapsto \mathit{generate}(\mathfrak{part}\langle \mathbf{If}, l_0, \mathbf{false}\rangle, d)$;*
- *the analysis of the body of both branches involve usual transfer functions;*
- *at point l_4 the join of the invariants corresponding to both branches should be computed, so that we get an invariant d_4, such that $\mathbf{tokens}_T\langle d_4\rangle = \{\mathfrak{part}\langle \mathbf{If}, l_0, \mathbf{true}\rangle :: \epsilon, \mathfrak{part}\langle \mathbf{If}, l_0, \mathbf{false}\rangle :: \epsilon\}$;*
- *at point l_5 the analyzer merges the partitions together, by applying the transfer functions $d \mapsto \mathit{merge}(\{\mathfrak{part}\langle \mathbf{If}, l_0, \mathbf{true}\rangle, \mathfrak{part}\langle \mathbf{If}, l_0, \mathbf{false}\rangle\}, d)$.*

5.3 Implementation and Experimental Evaluation

Last, we provide some details about the implementation of the partitioning domain, of its use in practice (i.e., the partitioning strategies) and the performances of the resulting

```
ℓ0 : s0;
ℓ1 : if(c){
ℓ2 :       s1
     }else{
ℓ3 :       s2
     }
ℓ4 : s3;
ℓ5 : s4;
```

(a) Initial program

```
ℓ0 : s0;
     ⟅Partition the traces in the following if statement⟆
ℓ1 : if(c){
ℓ2 :       s1
     }else{
ℓ3 :       s2
     }
ℓ4 : s3;
     ⟅Merge the partitions of the if statement at this point⟆
ℓ5 : s4;
```

(b) Program with directives added

Here are the main steps of the analysis:

Figure 5.5: Partitioning analysis of a **if**-statement: directives

analyzer.

5.3.1 Implementation of the domain

The data-structure: In practice, $\mathbf{tokens}_T\langle d_T\rangle$ can be considered the set of paths into the leaves of a tree, where each branch in the tree is labeled with a directive. Therefore, trees are a natural representation for the elements of $D^\sharp_{\mathbb{P},\mathbb{M}}$, with elements of $D^\sharp_{\mathbb{M}}$ at the leaves and with directives as labels for the branches:

Definition 5.3.1. Representation of the elements of $D^\sharp_{\mathbb{P},\mathbb{M}}$.

The physical representation of the elements of $D^\sharp_{\mathbb{P},\mathbb{M}}$ is defined by induction by:

$$\begin{array}{rcll} d_T & ::= & \mathrm{leaf}[d] \quad \textit{where } d \in D^\sharp_{\mathbb{M}} & \textit{(leaf } D^\sharp_{\mathbb{M}} \textit{ element)} \\ & | & \mathrm{node}[\phi] \quad \textit{where } \phi \in \mathcal{D} \to D^\sharp_{\mathbb{P},\mathbb{M}} & \textit{(function mapping directives into } D^\sharp_{\mathbb{P},\mathbb{M}}\textit{)} \end{array}$$

The use of this representation is exemplified in Example 5.3.1, after we define the transfer functions.

Remark 5.3.1. Use of the $\mathfrak{part}\langle\mathrm{None}\rangle$ directive.

In some cases, we may have to represent an invariant d_T, such that $\iota \in \mathbf{tokens}_T\langle d_T\rangle$ and $(\partial :: \iota) \in \mathbf{tokens}_T\langle d_T\rangle$ (for some token ι and some directive ∂). Then, the above definition does not provide a way to represent the invariant corresponding to ι since ι is a prefix of $\partial :: \iota$ and Definition 5.3.1 does not allow for numerical invariants to be assigned to nodes of the trees (numerical invariants correspond to leaves only).

The $\mathfrak{part}\langle\textbf{None}\rangle$ *directive solves this problem: indeed,* $\mathfrak{part}\langle\textbf{None}\rangle :: t$ *is equivalent to* t*, and a numerical invariant can be assigned to the leaf corresponding to* $\mathfrak{part}\langle\textbf{None}\rangle :: t$*. Such configurations do not occur in the analysis; they may arise in the invariant export (Section 5.1.3), when all local invariants corresponding to a control state* ℓ_0 *(possibly in different contexts, e.g., for different function calls) should be represented together. In particular the abstract join operator may generate* $\mathfrak{part}\langle\textbf{None}\rangle$ *directives.*

The transfer functions: The implementation of the transfer functions proceeds by induction on the structure of the trees. Indeed, let us consider the three kinds of transfer functions, which we introduced in Section 5.2.5 (in the following, we augment the names of the transfer functions for the partitioning domain with the index $_\mathbb{P}$):

- **Abstract binary operators, e.g. join** are defined by induction on the structure of trees.
  If the join of the set of paths in both trees contains two tokens t_0, t_1 such that t_0 is a strict prefix of t_1, then t_0 is replaced with $\mathfrak{part}\langle\textbf{None}\rangle :: t_0$ so that the result can be represented, as explained in Remark 5.3.1.
- **"Usual" transfer functions:** we consider the case of the $\mathit{guard}_\mathbb{P} : \mathbb{e} \times \mathbb{B} \times D^\sharp_{\mathbb{P},\mathbb{M}} \rightarrow D^\sharp_{\mathbb{P},\mathbb{M}}$ transfer function, which inputs a condition $e \in \mathbb{e}$, a boolean $b \in \mathbb{B}$, and an abstract element d and outputs an over-approximation of the stores in d which evaluate e into b (in the case of assignments, variable forget... are similar). The definition of $\mathit{guard}_\mathbb{P}$ is based on the function guard defined over $D^\sharp_\mathbb{M}$:

$$\forall e \in \mathbb{e},\ \forall b \in \mathbb{B},\ \left\{ \begin{array}{rcl} \mathit{guard}_\mathbb{P}(e, b, \mathrm{leaf}[d]) & = & \mathrm{leaf}[\mathit{guard}(e, b, d)] \\ \mathit{guard}_\mathbb{P}(e, b, \mathrm{node}[\phi]) & = & \mathrm{node}[\partial_p \mapsto \mathit{guard}(e, b, \phi(\partial_p))] \end{array} \right.$$

- **Partition creation:** the partition creation abstract transfer function $\mathit{generate} : \mathcal{D} \times D^\sharp_{\mathbb{P},\mathbb{M}} \rightarrow D^\sharp_{\mathbb{P},\mathbb{M}}$ inputs a partitioning directive ∂ and an abstract element d and pushes the token ∂ on top of the tokens. Basically, it mimics the creation of a partition triggered by the directive ∂, which amounts to adding a node on top of each leaf in d, with a branch indexed by ∂ in between:

$$\forall \partial \in \mathcal{D},\ \left\{ \begin{array}{rcl} \mathit{generate}(\partial, \mathrm{leaf}[d]) & = & \mathrm{node}[\partial \mapsto \mathrm{leaf}[d]] \\ \mathit{generate}(\partial, \mathrm{node}[\phi]) & = & \mathrm{node}[\partial_p \mapsto \mathit{generate}(\partial, \phi(\partial_p))] \end{array} \right.$$

  In practice, the partition generation function takes into account the names of the partitions, so as to create only *some* partitions.
- **Partition merge:** the transfer function $\mathit{merge} : \mathcal{P}(\mathcal{D}) \times D^\sharp_{\mathbb{P},\mathbb{M}} \rightarrow D^\sharp_{\mathbb{P},\mathbb{M}}$ inputs $D \subseteq \mathcal{D}, d \in D^\sharp_{\mathbb{P},\mathbb{M}}$; it goes recursively through the tree representing d and removes all occurrences of a directive in D. The implementation follows the following algorithm:

$$\forall D \in \mathcal{P}(\mathcal{D}),\ \left\{ \begin{array}{rcl} \mathit{merge}(D, \mathrm{leaf}[d]) & = & \mathrm{leaf}[d] \\ \mathit{merge}(D, \mathrm{node}[\phi]) & = & \mathrm{node}[\phi'] \\ \multicolumn{3}{l}{\quad \text{where } \phi' : \left\{ \begin{array}{lcl} \partial \notin D & \mapsto & \mathit{merge}(D, \phi(\partial)) \\ \mathfrak{part}\langle\textbf{None}\rangle & \mapsto & \bigsqcup\{d \text{ at a leaf of } \phi(\partial) \mid \partial \in D\} \end{array} \right.} \end{array} \right.$$

The directive $\mathfrak{part}\langle \textbf{None} \rangle$ allows to fold together *some* branches leaving from a node. In case all branches can be folded, then these directives can be safely removed from trees:

$$\text{node}[\{\mathfrak{part}\langle \textbf{None} \rangle \mapsto d_0\}] \to d_0$$

Definition 5.3.1. Application to the partitioning of an if-statement.
We consider the program considered in Example 5.2.1, with the partitioning strategy displayed in Figure 5.5(b). We assume that the analysis starts with a single partition (i.e., only one ongoing token at point ℓ_0).
Figure 5.6 displays the partitions obtained when the analysis reaches each control state in this program:

- *statement s_0 does not generate any new partition, so the layout of the abstract element for ℓ_1 (Figure 5.6(a)) is the same as for ℓ_0 (Figure 5.6(b));*
- *the conditional causes a partitioning of the traces at ℓ_1, so two trees are created after this point (yet, the partition corresponding to* **false** *is not created explicitly in the true branch, since it would be empty), which are depicted in Figure 5.6(c) and Figure 5.6(d);*
- *the abstract join outputs a new abstract element, with two partitions corresponding to both sides of the conditional at point ℓ_4 (Figure 5.6(e));*
- *the merge of partitions is performed after the analysis of s_3, so that the tree in ℓ_5 consists in only a leaf (Figure 5.6(f)) at in ℓ_0.*

As a shortcut, we write ∂_t for $\mathfrak{part}\langle \textbf{If}, \textbf{false}, \ell_1 \rangle$ and ∂_f for $\mathfrak{part}\langle \textbf{If}, \textbf{true}, \ell_1 \rangle$, and d for any invariant in $D^\sharp_{\mathbb{M}}$. Dotted lines denote the partitions which are not generated, since the analysis proves them empty.

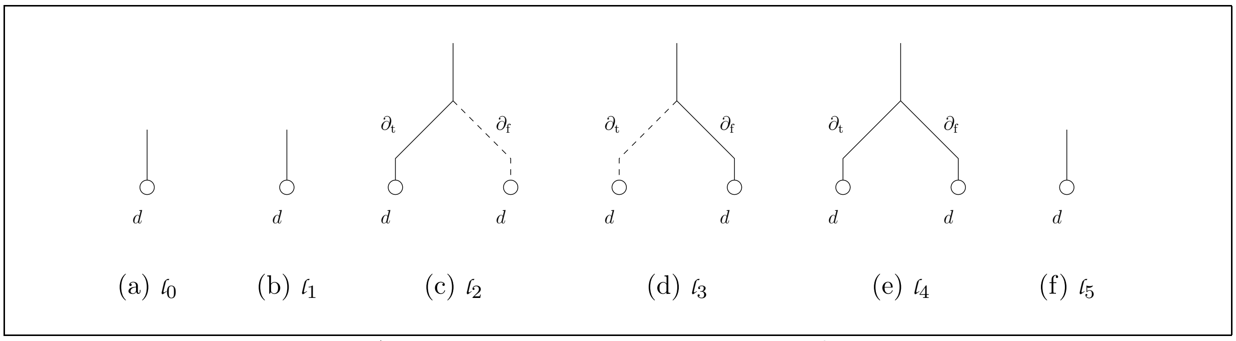


Figure 5.6: Application to the partitioning of an **if**-statement

5.3.2 Strategies for trace partitioning

Implementation of a partitioning strategy: As mentioned in Section 5.1.3, a preprocessing phase generates hints for the abstract domains, including the partitioning domain. Such hints specify the cases where partitions might be helpful in order to compute tighter invariants. In the analysis phase, partitioning may or may not be performed at these points, depending on the choice of the interpreter. Indeed, in case the pre-processing phase recommends a partitioning guided by the values of a variable v and the analyzer infers too large a range for v (i.e., the number of generated partitions would be prohibitive), the analyzer will not perform the partitioning. Similarly, it will not create empty partitions: for instance, in the case of a conditional statements which should be partitioned, if the analysis proves the condition always evaluates to **true**, then, the partition corresponding to the **false** branch will not be generated.

Strategies for generating "good" partitions: At the time we are writing this thesis, the design of the partitioning strategies was mostly done by Laurent Mauborgne. We enumerate a few cases where the current pre-processing phase suggests partitions to be generated:

- *sequences of conditional statements:* partitioning the traces in the first **if**-statement may greatly improve the precision in the following conditional statements, if the condition of the second **if**-statement depends on the content of the branches of the first one, or if its value depends on the value of the condition of the first **if**-statement.
- *assignment to an integer variable i* used *as an array index:* the partitioning guided by the value of i generates some relations with the variables in the right hand side of the assignment and may improve the precision of the subsequent array operation, since distinct array cells are treated separately, in a refined environment. This criterion causes the right partitions to be generated in the case of the interpolation function with regular discretization of the input, which we presented in Section 5.2.2, and Figure 5.3.
- *small loops assigning an integer variable i* used *e.g., as an array index:* the unrolling of the loop allows for the same kind of relations to be computed as in the previous point; hence, it results in the same opportunities for gains in precision. This criterion triggers the generation of the right partitions in the case of the interpolation function with indirection arrays, which we described in Section 5.2.2 and Figure 5.2.

5.3.3 Experimental results

This last subsection provides a few experimental data, which were collected when running the analyzer on several families of programs described in Section 5.1.1.

Methodology for the benchmarks: The results below were obtained on 2 GHz Bi-opteron machines, with 8 Gb of RAM (total) and 1 Mb of cache memory (per processor), running Linux. All the analyses reported below used only one processor, despite Astrée also features the ability of being ran in "*parallel*" mode.

The analyzer was ran on a series of programs, chosen among two families of embedded codes, which we detail in the table below. Programs in family 1 (denoted with $\mathcal{P}_i^1$) are older, and of smaller size than programs in family 2 (denoted with $\mathcal{P}_i^2$).

Program	Size (LOCs)	Functions	Variables			
			Global and static		Local	
			int	float	int	float
$\mathcal{P}_1^1$	370	20	23	87	2	0
$\mathcal{P}_2^1$	9 500	236	35 100	835	4	8
$\mathcal{P}_3^1$	70 000	2 010	11 700	27 400	22	516
$\mathcal{P}_1^2$	70 000	1 150	71 400	8 670	11 700	5 700
$\mathcal{P}_2^2$	226 000	3 410	35 700	24 900	44 300	21 900
$\mathcal{P}_3^2$	400 000	5 680	58 700	35 500	83 400	35 100

Partitioning strategy: The following table displays the results of the partitioning strategy. We give the total number of conditional structures, and the number of *partitioned* conditional structures. We provide similar information about the partitioning of loop structures; however, only the internal loops are taken into account here (we recall that a program in either families consists in a main loop, which contains most of the code). Last, we mention the number of directives recommending a partitioning guided by the values of a variable.

Program	Size (LOCs)	Conditional		Loops		Value-based partitioning
		partitioned	total	partitioned	total	
$\mathcal{P}_1^1$	370	4	28	1	1	0
$\mathcal{P}_2^1$	9 500	18	283	1	3	0
$\mathcal{P}_3^1$	70 000	498	4617	3	5	112
$\mathcal{P}_1^2$	70 000	300	2624	106	106	0
$\mathcal{P}_2^2$	226 000	1805	9381	591	591	19
$\mathcal{P}_3^2$	400 000	2802	17562	906	916	32

Overall, partitioning directives are inserted in the case of 10 % to 20 % of the conditional structures and for almost all internal loops. The partitioning guided by the values of variables tend to have less importance (much fewer directives inserted, and only in the larger applications).

Analysis with partitioning enabled: In the following T.p.I. stands for "Time per iteration"; it corresponds to the average time spent in *one* iteration of the main loop of the program being analyzed. This time is roughly representative of the efficiency of the transfer functions and of the precision of the abstract control flow. The number of iterations assesses the efficiency of the convergence. The global time of the analysis depends both on the efficiency of transfer functions and the speed of the convergence.

Times are written in seconds (s); amounts of memory in megabytes (Mb).

The first benchmark displays the result of the analysis with the default settings: *trace partitioning is enabled* and the directives are inserted by the *automatic strategy*, evoked in Section 5.3.2.

	Size	Memory peak (Mb)	Analysis time (s)	Iterations	T.p.I. (s)	False alarms
$\mathcal{P}_1^1$	370	45	1.96	9	0.21	0
$\mathcal{P}_2^1$	9 500	175	104	17	6.1	8
$\mathcal{P}_3^1$	70 000	636	2 818	35	80.5	0
$\mathcal{P}_1^2$	70 000	434	1 064	20	53.2	0
$\mathcal{P}_2^2$	226 000	1 533	17 035	51	334	0
$\mathcal{P}_3^2$	400 000	2 423	36 480	72	507	0

Global impact of partitioning: First, we compare the results of the analyses with or without trace partitioning enabled: the table below displays the results *without* trace partitioning. Note that the partitioning inherent in the function calls (function inlining) is not affected by the disabling of trace partitioning: turning off partitioning removes the partitioning relative to loop iterations, conditional and variables values only.

The number in parentheses allow to compare with the default, partitioning analyses.

	Size	Memory peak	Analysis time	Iterations, T.p.I.	Alarms
$\mathcal{P}_1^1$	370	45 (-)	1.55 (-21 %)	9, 0.17s	0 (0)
$\mathcal{P}_2^1$	9 500	170 (- 3 %)	87 (- 17 %)	17, 5.1s	8 (8)
$\mathcal{P}_3^1$	70 000	660 (+ 3 %)	1 614 (- 43 %)	35, 46.1s	750 (0)
$\mathcal{P}_1^2$	70 000	376 (-13 %)	921 (- 13 %)	20, 46s	443 (0)
$\mathcal{P}_2^2$	226 000	1 341 (- 12 %)	37 274 (+ 112 %)	282, 134s	5 402 (0)
$\mathcal{P}_3^2$	400 000	2 040 (- 16 %)	34 147 (- 6 %)	127, 269s	7 524 (0)

This first comparison shows the great impact of partitioning in most cases, and especially in the case of the large applications, i.e., the programs which compare most closely with real applications due to their size and structure. The first two programs are experimental programs, which do not comprise all the features of the largest applications and involve smaller chains of computations, so the trace partitioning does not impact the number of alarms. Yet, the invariants are noticeably less precise, even in the case of the first example. The analyses of larger, real-world applications generate dramatic number of alarms: trace partitioning proves a crucial technique in ASTRÉE.

Secondly, we remark that the execution time is not necessarily better when trace partitioning is disabled. In particular, the analysis of the two largest programs require a *much larger* number of iterations when trace partitioning is turned off: this effect was most noticeable in the case of the second program in the second family (282 iterations instead of 52!). In fact, a lower precision *may* result in a longer analysis time for many reasons related to the exploration of a larger state space:

- the widening of the analyzer attempts to stabilize variables, with a widening threshold scale [BCC$^+$03a]; therefore, if some variable cannot be stabilized to a small range (for instance, because some property cannot be proved due to the trace partitioning being turned off), it goes through a longer sequence of widened ranges (the analyzer attempts to find a larger, stable range), before it eventually reaches the "top" value (i.e., range containing all concrete values). This is an explanation for larger numbers of iterations in the case of less precise analyses.
- the control flow of the static analysis need to be more exhaustive when the precision is worse: for instance, in the case of a conditional, a less precise input invariant may require the analysis of *both* branches of the conditional whereas a more precise invariant may require analyzing only one branch, hence, require less time to complete.

Overall, we remark that the time per iteration is lower in the case of non-partitioning analyses and the partitioning analyses tend to require a lower number of iterations However, it is difficult to say for sure what is the most important factor: we may guess that only the first factor plays a significant role here (longer analyses due to longer widening chains), however, we should remark that the non-partitioning transfer functions handle much simpler data-structures; the latter factor may explain the shorter iterations.

Moreover, it is rather intuitive that one iteration of a partitioning analysis should take longer than one iteration of a non-partitioning analysis; however, the cost in time of trace partitioning (whether global analysis time or time per iteration) never turns out prohibitive.

Last, we remark that partitioning analyses require more memory in most cases; this result is to be expected, since partitioning analyses generate more data-structures and handle more numerical invariants. Yet, this cost is rather reasonable, since it never goes above 20 % (10 % average). This is mostly due to the fact that most partitioning criteria are *local*: they do not yield to huge sets of global partitions, thanks to the insertion of merge directives (Section 5.2.1).

In the following, we focus on several kinds of partitioning criteria and measure their impact on the results of the analysis.

Impact of the partitioning of conditional structures: Second, we compare the default, partitioning analysis with analyses carried out without *some* partitions. The table below reports the result of the analysis without partitioning of conditional structures.

	Size	Memory peak	Analysis time	Iterations, T.p.I.	Alarms
$\mathcal{P}_1^1$	370	45 (-)	1.96 (-)	9, 0.17s	0 (-)
$\mathcal{P}_2^1$	9 500	173 (- 1 %)	88 (- 15 %)	17, 5.2s	8 (-)
$\mathcal{P}_3^1$	70 000	616 (- 3 %)	5 004 (+ 76 %)	32, 156s	398 (0)
$\mathcal{P}_1^2$	70 000	467 (+ 8 %)	1 466 (+ 38 %)	20, 73.2s	389 (0)
$\mathcal{P}_2^2$	226 000	1 680 (+ 10 %)	199 500 (+ 1071 %)	290, 688s	5 190 (0)
$\mathcal{P}_3^2$	400 000	2 735 (+ 12 %)	187 773 (+ 415 %)	125, 1502s	5 542 (0)

The results in precision fall between the results of the partitioning analysis and the results of the non-partitioning analysis. In the case of the largest applications, the number of alarms is still dramatic.

In the resource usage point of view, these results are much worse than those of the non-partitioning analysis and of the partitioning analysis. Not only the number of iterations but also the time per iteration tend to be worse than those of the partitioning analysis (despite simpler structures being used). At this point, we can imagine that not only the disabling of the partitioning of **if**-statements caused the analyzer to go through longer widening chains but also that it resulted in a coarser approximation of control flow. Another possibility is that the imprecision due to the absence of partitioning after **if**-statement may cause more imprecise partitions based on other criteria (loops, values of variables) to be generated, resulting in worse performances.

Inner loops partitioning: The table below reports the result of the analysis without partitioning of loops.

	Size	Memory peak	Analysis time	Iterations, T.p.I.	Alarms
$\mathcal{P}_1^1$	370	45	1.96 (-)	9, 0.21s	0 (-)
$\mathcal{P}_2^1$	9 500	173 (-1 %)	85 (-18 %)	17, 5s	8 (-)
$\mathcal{P}_3^1$	70 000	596 (- 6 %)	3 928 (+ 39 %)	63, 62.3s	529 (0)
$\mathcal{P}_1^2$	70 000	391 (- 10 %)	12 319 (+1 058 %)	292, 42.2s	208 (0)
$\mathcal{P}_2^2$	226 000	1 400 (- 9 %)	14 277 (- 16 %)	75, 190s	2 954 (0)
$\mathcal{P}_3^2$	400 000	2 204 (- 9 %)	41 932 (+ 15 %)	115, 364s	4 017 (0)

Again, we remark that loop partitioning is crucial for the precision of the analyses in the case of large applications, since the analysis of the four larger applications generate hundreds or thousands of false alarms. The invariants generated for the other programs are also significantly less precise (even though, the imprecision does not cause a larger number of alarms).

In the analysis time point of view, the same comments as above apply: in general the number of iterations is bigger, the time per iteration is smaller. In some cases ($\mathcal{P}_2^2$), the analysis is faster; in other cases ($\mathcal{P}_3^1, \mathcal{P}_1^2, \mathcal{P}_3^2$) it is slower. We note that $\mathcal{P}_1^2$ requires a very large number of iterations.

Impact of value-guided partitioning:

	Size	Memory peak	Analysis time	Iterations, T.p.I.	Alarms
$\mathcal{P}_1^1$	370	45 (-)	1.58 (- 27 %)	9, 0.18s	0 (-)
$\mathcal{P}_2^1$	9 500	173 (-)	82 (- 20 %)	17, 4.8s	8 (8)
$\mathcal{P}_3^1$	70 000	682 (+ 7 %)	2 236 (+ 26 %)	33, 67.8s	563 (0)
$\mathcal{P}_1^2$	70 000	438 (+ 1 %)	1 335 (+ 25 %)	20, 66.7s	4 (0)
$\mathcal{P}_2^2$	226 000	1 550 (+ 1 %)	16 589 (- 3 %)	66, 251s	3 (0)
$\mathcal{P}_3^2$	400 000	2 434 (-)	26 165 (- 28 %)	64, 409s	8 (0)

The impact of partitioning guided by values is less significant than the impact of the previous partitioning criteria, except in the case of the program $\mathcal{P}_3^1$ (dramatic number of alarms).

We report no very important difference in execution time. Yet, we note that the more precise analysis of $\mathcal{P}_2^2$ requires *more* iterations.

Overall, it turns out extremely difficult to explain all variations in resources required by static analyses: no rule allows to predict the speed of an analysis; and, in practice, too many factors play a role, even though one may be able to tell in some cases what the most important ones are.

5.3.4 Related work

As a conclusion of this chapter about the implementation of trace partitioning in the Astrée static analyzer, we provide some data about related work.

We can find several occurrences of refinements of the control structure in the literature about data-flow analysis. For instance, [SP81] studied the most common approaches to interprocedural analyses. A finer handling of paths in control flow graphs was proposed in [HR80]: it proceeds by integrating some information about the paths in the edges of the control flow graph, so as to allow for a finer approximation of the control flow to be computed. In particular, this technique was used in order to infer sets of *feasible paths*, so as to allow for more precise data-flow analyses. Similarly [BGS97] determines branch correlations so as to detect incompatible branchings and cut down the approximation set of feasible paths. Our approach not only performs intuitive abstractions of the paths, but also takes the path into account dynamically during the analysis.

The qualified flow analysis technique was extended with path profiles [BL96] in [AL98]: profiling data should determine a set of *hot* paths (i.e., more frequently taken); then, these paths can be analyzed separately, with a higher precision (no path joins). Similarly, the express lane transformation [MR03] aims at duplicating hot paths, so as to improve precision. However, this approach does not apply in our case. First, profiling very large applications with very large numbers of variables does not seem a realistic solution (at least in the time point of view). Secondly, this approach analyzes all "non-hot paths" together (i.e., with no partitioning), which would result in a low precision, with possibly many alarms. Indeed, the precision required in the analysis of a path for proving it safe is not related to how frequently it is used; therefore, our approach ignoring the frequency of paths is more adapted to program certification.

A trace partitioning static analysis framework was proposed in [HT98]; however, this framework does not allow for the *merge* of partitions. Therefore, it incurs an exponential cost (in the number of **if** and **while** statements). Moreover, it does not allow for the dynamic partitioning guided by the values of a variable.

Recently, a large number of path sensitive analyses were proposed and implemented in various frameworks, such as [BR01, FLL$^+$02] and contributed to the verification of complex properties. However, path sensitivity is very costly in practice: we could not apply this technique to a single iteration of the main loop of either of the programs considered in Section 5.3.3. An interesting solution to the cost of path sensitivity (yet, not applicable in our case) proposed in [DLS02] relies on the encoding of the property of interest into an automaton (*finite state machine*): the transitions in the automaton can be used as criteria for partitioning the paths, and a heuristic is introduced so as to merge paths as well.

Chapter 6

Partitioning and Synchronous Product

We propose a second instantiation for the trace partitioning framework, which we set up in Chapter 4. The purpose of this instantiation is to partition traces according to an abstraction of the history of program executions defined by a collection of "events".

This approach should allow to discriminate traces which satisfy some conditions defined from the history of program executions (such as: condition P was satisfied at point l_0 at the previous iteration in a loop and is violated at the current iteration) prove some functional properties of programs.

We define a collecting semantics for expressing these properties in Section 6.1; then, we set up a framework for defining generic abstractions of this collecting semantics in Section 6.2, in order to derive some decidable approximation of it: Section 6.3 specializes it with automata. Section 6.4 specializes it with numerical domains.

6.1 The Partitioning

6.1.1 Motivation for a new instantiation of the trace partitioning framework

We proposed a framework for partitioning traces in Chapter 4 and a first instantiation of it in Chapter 5, so as to improve the precision of static analyzers (the approach was integrated into ASTRÉE and contributed to the precision and efficiency of the analyses). The purpose of this chapter is to provide a second instantiation of the trace partitioning framework.

We wish to use partitions of traces in order to:

- **discriminate sets of executions** satisfying certain properties, such as "some event occurred an even number of times" or "some property occurs during the *next* iteration of some loop". These properties cannot be expressed by the instantiation of the partitioning framework proposed in Chapter 5, yet they clearly amounts to partitions

of the traces in the sense of Chapter 4. In particular, the constructions presented in this Chapter will be thoroughly used in the definition of semantic slicing, which we introduce in Chapter 7.
- **integrate the properties of interest in the static analysis** so as to let the analyzer make more sensible abstractions, and allow the proof to succeed. This goal was secondary, when we developed the abstractions mentioned in this chapter. However, it seems that these abstraction could play a great role in the verification of simple *functional properties* of programs by ASTRÉE.

The trace partitioning framework of Chapter 4 is most adapted to the definition of such a collecting semantics and of such abstractions.

The approach proposed in this chapter is related to the synchronous product of the program to analyze with an adapted control structure: this method has been proposed and widely used for the verification of synchronous programs [HLR93]. For instance, [Jea03] carries out a partitioning of the boolean control structure of a product of Lustre programs with their specification (implemented as a monitor), so as to check that the programs abide by the specifications.

This method is most popular in model checking [EMCP02]; it allows to take the property of interest to be taken into account during the model refinement phase and the model checking phase.

The purpose of this chapter is two folds:
- we wish to integrate these methods into an existing analyzer; the definition of a trace partitioning domain turns out an efficient solution towards that goal;
- we intend to set up a *collecting semantics*, which takes into account a broad family of "monitors"; the choice of the monitor amounts to choosing an abstraction of sets of tokens (moreover, this approach allows "abstract" monitors to be defined, hence, allows for more flexibility in the analyses).

6.1.2 Language extension

We first extend the syntax of the simple language defined in Section 2.2 with a new statement, called **cnt**. Intuitively, the **cnt** statements count the number of times they are executed, and remember in which order.

First, we define the syntax and *standard* semantics of this new kind of statement, in the same way as in Section 2.2.2; this standard semantics basically ignores the **cnt** statements. Indeed, keeping track of the execution of **cnt** statements requires carrying some *tokens* in the sense of Section 4.2, so that Section 6.1.3 defines a *non standard* semantics, so as to keep track of the execution of the **cnt** statements; it will be based on extended systems.

Definition 6.1.1. cnt-statement.

The syntax of the **cnt***-statement is:* l : **cnt**; l' : . . . .

The standard semantics of a **cnt***-statement is the same as the semantics of a skip state-*

*ment. For instance, in the case of the statement ℓ : **cnt**; ℓ' : ...*

$$\forall \rho \in \mathbb{M},\ (\ell, \rho) \rightarrow (\ell', \rho)$$

6.1.3 Semantic extension

As mentioned in Section 6.1.2, the semantics of the **cnt**-statement should keep track of the number of times they are executed and in which order. This requires some extension of the semantics to be defined, which amounts to choosing a suitable extended system, as defined in Definition 4.2.2.

We assume that a program P is chosen. First, we define a set of extended tokens:

Definition 6.1.2. Tokens.

The set of directives is $\mathcal{D} = \{\partial_\ell \mid \ell \in \mathbb{L}\}$.
The set $\mathbb{T}$ of tokens is made of stacks of tokens, hence is generated by the following grammar:

$$\begin{array}{rcll} t(t \in \mathbb{T}) & ::= & \epsilon & \textit{(empty stack, initial partition)} \\ & \mid & \partial_\ell :: t & \textit{where } \ell \in \mathbb{L},\ t \in \mathbb{T} \textit{ (push, stack)} \end{array}$$

Intuitively, a directive corresponds to the control state of a **cnt** statement, and a token collects the list of the **cnt** statements in the order they are executed in: ϵ stands for the empty list (initial configuration); the token $\partial_\ell :: t$ is generated after running a ℓ: **cnt**; statement, from the configuration t. The following definition sets up the corresponding extended transition system:

Definition 6.1.3. Extended system.

The transition relation of the extended system is defined by:

- *for the counter statement ℓ : **cnt**; ℓ':*

$$\forall \rho \in \mathbb{M},\ t \in \mathbb{T},\ ((\ell, t), \rho) \rightarrow_{\mathbb{T}} ((\ell', \partial_\ell :: t), \rho)$$

- *for any other transition in the original system, if $(\ell, \rho) \rightarrow (\ell', \rho')$, then for any token $t \in \mathbb{T}$, $((\ell, t), \rho) \rightarrow_{\mathbb{T}} ((\ell', t), \rho')$.*

The initial states of the extended system is $\mathbb{L}^{\mathrm{i}}_{\mathbb{T}} = (\ell^{\mathrm{i}}, \epsilon)$.
We write $P_{\mathbb{T}}$ for this extended system.

The following remark is straightforward:

Theorem 6.1.1. A complete partition.

The system $P_{\mathbb{T}}$ is a τ-complete partition of the trivial extension P_ϵ of the initial program P, where $\tau : t \mapsto t_\epsilon$.

Proof.
The properties listed in Definition 4.2.2 can be established straightforwardly. □

Definition 6.1.1. Infinite loop, with cnt-statement.
Let us consider the following infinite loop:

$$\begin{array}{ll}
\ell_0: & \text{bool } b;\\
\ell_1: & \textbf{while}(\textbf{true})\{\\
\ell_2: & \quad \textbf{cnt};\\
\ell_3: & \quad \textbf{input}(b);\\
\ell_4: & \quad \textbf{if}(b)\{\\
\ell_5: & \quad\quad \textbf{cnt};\\
\ell_6: & \quad \}\\
\ell_7: & \}\\
\ell_8: & \ldots
\end{array}$$

We give fragments of some traces of both programs in the table below (we show only the control states, and abstract the stores away for the sake of concision); note that σ_i stands for a trace of the initial system, whereas σ_i' denotes the corresponding trace in the extended system:

initial program P	*extended system $P_{\mathbb{T}}$*
$\sigma_1 = \langle \ell_0, \ell_1, \ell_2, \ell_3, \ell_4, \ell_7, \ell_1, \ell_2, \ell_3, \ell_4, \ell_7 \rangle$	$\sigma_1' = \langle (\ell_0, \epsilon), (\ell_1, \epsilon), (\ell_2, \epsilon), (\ell_3, \partial_{\ell_2} :: \epsilon), (\ell_4, \partial_{\ell_2} :: \epsilon), (\ell_7, \partial_{\ell_2} :: \epsilon), (\ell_1, \partial_{\ell_2} :: \epsilon), (\ell_2, \partial_{\ell_2} :: \epsilon), (\ell_3, \partial_{\ell_2} :: \partial_{\ell_2} :: \epsilon), (\ell_4, \partial_{\ell_2} :: \partial_{\ell_2} :: \epsilon), (\ell_7, \partial_{\ell_2} :: \partial_{\ell_2} :: \epsilon) \rangle$
$\sigma_2 = \langle \ell_0, \ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6, \ell_7, \ell_1, \ell_2, \ell_3, \ell_4, \ell_7 \rangle$	$\sigma_2' = \langle (\ell_0, \epsilon), (\ell_1, \epsilon), (\ell_2, \epsilon), (\ell_3, \partial_{\ell_2} :: \epsilon), (\ell_4, \partial_{\ell_2} :: \epsilon), (\ell_5, \partial_{\ell_2} :: \epsilon), (\ell_6, \partial_{\ell_5} :: \partial_{\ell_2} :: \epsilon), (\ell_7, \partial_{\ell_5} :: \partial_{\ell_2} :: \epsilon), (\ell_1, \partial_{\ell_5} :: \partial_{\ell_2} :: \epsilon), (\ell_2, \partial_{\ell_5} :: \partial_{\ell_2} :: \epsilon), (\ell_3, \partial_{\ell_2} :: \partial_{\ell_5} :: \partial_{\ell_2} :: \epsilon), (\ell_4, \partial_{\ell_2} :: \partial_{\ell_5} :: \partial_{\ell_2} :: \epsilon), (\ell_7, \partial_{\ell_2} :: \partial_{\ell_5} :: \partial_{\ell_2} :: \epsilon) \rangle$
$\sigma_3 = \langle \ell_0, \ell_1, \ell_2, \ell_3, \ell_4, \ell_7, \ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6 \rangle$	$\sigma_3' = \langle (\ell_0, \epsilon), (\ell_1, \epsilon), (\ell_2, \epsilon), (\ell_3, \partial_{\ell_2} :: \epsilon), (\ell_4, \partial_{\ell_2} :: \epsilon), (\ell_7, \partial_{\ell_2} :: \epsilon), (\ell_1, \partial_{\ell_2} :: \epsilon), (\ell_2, \partial_{\ell_2} :: \epsilon), (\ell_3, \partial_{\ell_2} :: \partial_{\ell_2} :: \epsilon), (\ell_4, \partial_{\ell_2} :: \partial_{\ell_2} :: \epsilon), (\ell_5, \partial_{\ell_2} :: \partial_{\ell_2} :: \epsilon), (\ell_6, \partial_{\ell_5} :: \partial_{\ell_2} :: \partial_{\ell_2} :: \epsilon) \rangle$

These three examples show that the tokens retain the order and the control states corresponding to the **cnt** *statements which were executed:*

- *in the case of σ_1, the branch of the conditional is taken neither in the first iteration nor in the second: hence, ∂_{ℓ_5} does* not *appear in σ_1';*
- *in the case of σ_2, the conditional is ran in the first iteration and not in the second: hence, ∂_{ℓ_5} appears in σ_2' after the first occurrence of ∂_{ℓ_2} only;*
- *in the case of σ_3, the conditional is ran in the second iteration only: hence, ∂_{ℓ_5} appears in σ_2' after the second occurrence of ∂_{ℓ_2} only.*

Clearly, the number of tokens which may appear in the semantics of the extended system is infinite. However, the static analyses defined in Section 4.3 require the number of tokens to be *finite*, or at least that only a finite number of tokens is generated during the analysis. Therefore, we propose to design abstractions for these tokens defined as sequences of directives.

6.2 Abstractions of the concrete extension

6.2.1 Abstractions of the extension

We introduce in this section an abstraction for the extended system presented in Section 6.1.3, which proceeds by abstracting the sets of tokens $\mathbb{T}$ introduced in Section 6.1.2.

Definition 6.2.1. Token abstraction.

A token abstraction is defined by a set $\mathbb{T}^\sharp$ and a function $\gamma_\mathbb{T} : \mathbb{T}^\sharp \to \mathcal{P}(\mathbb{T})$. A element of $\mathbb{T}^\sharp$ is called an abstract token*; the function $\gamma_\mathbb{T}$ is called* token concretization.

A token forget relation can be defined from $\gamma_\mathbb{T}$, in the like of $\Rightarrow^{\mathbb{L}}_{\tau}$ in Remark 4.2.1. We write $(\Rightarrow_{\gamma_\mathbb{T}}) \subseteq (\mathbb{T} \times t^\sharp)$ for this relation; it is defined by:

$$\forall t^\sharp \in \mathbb{T}^\sharp,\ \forall t \in \mathbb{T},\quad (t \Rightarrow_{\gamma_\mathbb{T}} t^\sharp) \iff (t \in \gamma_\mathbb{T}(t^\sharp))$$

In case the abstract tokens form a partition of $\mathcal{P}(\mathbb{T})$, then, for all $t \in \mathbb{T}$ there exists a unique $t^\sharp \in \mathbb{T}^\sharp$ such that $t \in \gamma_\mathbb{T}(t^\sharp)$ so the relation $\Rightarrow_{\gamma_\mathbb{T}}$ can be turned into a function.

Given an abstraction for tokens, we can design an abstract extended system as follows:

Definition 6.2.2. Abstract extended system.

An abstract extended system *is an extended system $P_{\mathbb{T}^\sharp} = (\mathbb{L}_{\mathbb{T}^\sharp}, \mathbb{S}^{\mathrm{i}}_{\mathbb{T}^\sharp}, \to_{\mathbb{T}^\sharp})$ using abstract tokens and such that $P_{\mathbb{T}^\sharp}$ is a $\Rightarrow_{\gamma_\mathbb{T}}$-*covering *of $P_\mathbb{T}$, in the sense of Remark 4.2.1:*

- $\forall s \in \mathbb{S}^{\mathrm{i}}_\mathbb{T},\ \exists s' \in \mathbb{S}^{\mathrm{i}}_{\mathbb{T}^\sharp},\ s' \Rightarrow_{\gamma_\mathbb{T}} s;$
- $\forall s_0, s_1 \in \mathbb{L}_\mathbb{T},\ s'_0 \in \mathbb{T}^\sharp,\ (s_0 \Rightarrow_{\gamma_\mathbb{T}} s'_0 \wedge s_0 \to_\mathbb{T} s_1) \Longrightarrow \exists s_1,\ \begin{cases} s_1 \Rightarrow_{\gamma_\mathbb{T}} s'_1 \\ s'_0 \to_{\mathbb{T}^\sharp} s'_1 \end{cases}$

Clearly, the abstract extended system can be systematically derived from the extended system $P_\mathbb{T}$ and from the definition $(\mathbb{T}^\sharp, \gamma_\mathbb{T})$ of the token abstraction. As noted above, in case $\mathbb{T}^\sharp$ forms a partition of $\mathcal{P}(\mathbb{T})$, $\Rightarrow_{\gamma_\mathbb{T}}$ can be turned into a function, so that the above definition can be based on the standard notion of covering, defined by a function (Definition 4.2.2).

Overall, our approach proceeds by a two steps extension of the original system, as depicted in the diagram below:

1. extension of P into the *complete partition* $P_\mathbb{T}$, so as to define the non-standard semantics;

2. choice of an abstraction of $P_{\mathbb{T}}$, defined by an abstraction $(\mathbb{T}^\sharp, \gamma_{\mathbb{T}})$ of $\mathbb{T}$, so as to abstract the non-standard semantics.

$$\begin{tikzcd}
 & & P_{\mathbb{T}_1^\sharp} \\
P \arrow[r, mapsto, "\text{non standard semantics}"] & P_{\mathbb{T}} \arrow[ur, "\mathbb{T}_1^\sharp"] \arrow[r, "\mathbb{T}_2^\sharp"'] \arrow[dr] & P_{\mathbb{T}_2^\sharp} \\
 & & \ldots
\end{tikzcd}$$

Theorem 6.2.1. Abstract extended systems as coverings.

An abstract extended system $P_{\mathbb{T}^\sharp}$ is a covering of the initial system P, with respect to the trivial projection function.

Proof.

The transitivity result in Theorem 4.2.5 applies. □

We now present an abstraction of the extended system defined in Example 6.1.1:

Definition 6.2.1. Abstraction.

We write **occurences**$(\!|\partial \in \mathsf{t}|\!)$ *for the number of occurrences of ∂ in t.*

We propose a very simple abstraction of tokens, with two abstract values:

- $\mathsf{t}_=^\sharp$ *stands for the tokens where the number of occurrences of ∂_{ℓ_2} is the same as the number of ∂_{ℓ_5};*
- $\mathsf{t}_<^\sharp$ *stands for the tokens where the number of occurrences of ∂_{ℓ_5} is smaller than the number of ∂_{ℓ_2} ($\mathsf{t}_>^\sharp$ is defined similarly).*

Formally,

$$\begin{array}{rlcl}
\gamma_{\mathbb{T}}: & \mathsf{t}_<^\sharp & \mapsto & \{\mathsf{t} \mid \mathbf{occurences}(\!|\partial_{\ell_5} \in \mathsf{t}|\!) < \mathbf{occurences}(\!|\partial_{\ell_2} \in \mathsf{t}|\!)\} \\
 & \mathsf{t}_=^\sharp & \mapsto & \{\mathsf{t} \mid \mathbf{occurences}(\!|\partial_{\ell_5} \in \mathsf{t}|\!) = \mathbf{occurences}(\!|\partial_{\ell_2} \in \mathsf{t}|\!)\} \\
 & \mathsf{t}_>^\sharp & \mapsto & \{\mathsf{t} \mid \mathbf{occurences}(\!|\partial_{\ell_5} \in \mathsf{t}|\!) > \mathbf{occurences}(\!|\partial_{\ell_2} \in \mathsf{t}|\!)\}
\end{array}$$

Basically, the elements in the partition corresponding to the token $\mathsf{t}_=^\sharp$ are such that the **true** *branch in the conditional is* always *taken, whatever the iteration number.*

Then, the abstract extended system is defined by:

- *in the beginning, the number of the execution,* **occurences**$(\!|\partial_{\ell_2} \in \mathsf{t}|\!) =$ **occurences**$(\!|\partial_{\ell_5} \in \mathsf{t}|\!)$*, since $\mathsf{t} = \mathsf{t}_\epsilon$ in the beginning of the execution (neither ∂_{ℓ_2} nor ∂_{ℓ_5} have been encountered yet);*
- *the transitions related to the statement ℓ_5 :* **cnt**; ℓ_6 *are the following:*

$$\forall \rho \in \mathsf{M}, \left\{ \begin{array}{lcl}
((\ell_5, \mathsf{t}_<^\sharp), \rho) & \rightarrow_{\mathbb{T}^\sharp} & ((\ell_6, \mathsf{t}_<^\sharp), \rho) \\
((\ell_5, \mathsf{t}_<^\sharp), \rho) & \rightarrow_{\mathbb{T}^\sharp} & ((\ell_6, \mathsf{t}_=^\sharp), \rho) \\
((\ell_5, \mathsf{t}_=^\sharp), \rho) & \rightarrow_{\mathbb{T}^\sharp} & ((\ell_6, \mathsf{t}_>^\sharp), \rho) \\
((\ell_5, \mathsf{t}_>^\sharp), \rho) & \rightarrow_{\mathbb{T}^\sharp} & ((\ell_6, \mathsf{t}_>^\sharp), \rho)
\end{array} \right.$$

The transitions defined by the other **cnt** *statement are similar. The other transitions can be derived straightforwardly from the standard semantics (Section 2.2.2).*

We can note that the abstract tokens form a partition of the set of concrete tokens, so that the abstract extended system is a covering of $P_{\mathbb{T}}$ defined by a function instead of a mere relation $\Rightarrow_{\gamma_{\mathbb{T}}}$.

6.2.2 Design of the interpreter

We now propose to extend the abstract interpreter defined in Section 3.2.5.

Abstract operations: The definition of such an interpreter requires $\mathbb{T}^\sharp$ to provide an initial abstract token and an abstract counterpart for the operation which adds a directive on top of a token.

Definition 6.2.3. Abstract initial token.

An abstract initial token *is an element $t_\epsilon^\sharp \in \mathbb{T}^\sharp$ such that $\epsilon \in \gamma_{\mathbb{T}}(t_\epsilon^\sharp)$.*

Definition 6.2.4. Abstract push operation.

An abstract push operation *is a function $\mathit{push} : \mathbb{T}^\sharp \times \mathcal{D} \to \mathcal{P}(\mathbb{T}^\sharp)$, such that:*

$$\forall t_0^\sharp \in \mathbb{T}^\sharp,\ \partial \in \mathcal{D},\ t \in \gamma_{\mathbb{T}}(t_0^\sharp),\ \exists t_1^\sharp \in \mathit{push}(t_0^\sharp, \partial),\ (\partial :: t) \in \gamma_{\mathbb{T}}(t_1^\sharp)$$

We exemplify the above definitions in the case of Example 6.2.1:

Definition 6.2.2. Abstract initial token and push operation.

We let:

- $t_\epsilon^\sharp = t_=^\sharp$ *be the abstract initial token;*
- push *be defined by:*

$$\begin{array}{rlcl}
\mathit{push}: & (t_<^\sharp, \partial_{l_2}) & \mapsto & \{t_<^\sharp\} \\
& (t_<^\sharp, \partial_{l_5}) & \mapsto & \{t_<^\sharp, t_=^\sharp\} \\
& (t_=^\sharp, \partial_{l_2}) & \mapsto & \{t_<^\sharp\} \\
& (t_=^\sharp, \partial_{l_5}) & \mapsto & \{t_>^\sharp\} \\
& (t_>^\sharp, \partial_{l_2}) & \mapsto & \{t_>^\sharp, t_=^\sharp\} \\
& (t_>^\sharp, \partial_{l_5}) & \mapsto & \{t_>^\sharp\}
\end{array}$$

Partitioning, forward abstract interpreter: In the same way as in Section 3.2.5, Section 4.3.4 and Section 5.2.4, we define an abstract interpreter in denotational style, by induction on the syntax of programs. The abstract interpretation of a statement s should be defined as usual by a function $[\![s]\!]^\sharp : D^\sharp_{\mathbb{P},\mathbb{M}} \rightarrow D^\sharp_{\mathbb{P},\mathbb{M}}$, satisfying the usual soundness condition that it approximates the behavior of the statement.

We treat the case of a few statements:

- the most interesting case is of the **cnt**-statement, which should recompute partitions, by applying the push function to abstract tokens:

$$[\![l : \mathbf{cnt}]\!]^\sharp : d \mapsto \lambda(t_0^\sharp \in \mathbb{T}^\sharp) \cdot \bigsqcup\{d(t_1^\sharp) \mid t_0^\sharp \in \mathit{push}(t_1^\sharp, \partial_l)\}$$

- the case of the other statements is rather straightforward, since they do not affect the abstract partitions; for instance, in the case of the assignment:

$$[\![x := e]\!]^\sharp : d \mapsto \lambda(t^\sharp \in \mathbb{T}^\sharp) \cdot \mathit{assign}(x, e, d(t^\sharp))$$

  Basically, the semantics of other statements proceeds by a pointwise extension of the abstract operations defined in $D^\sharp_{\mathbb{M}}$.

An abstract join operator in $D^\sharp_{\mathbb{P},\mathbb{M}}$ can also be defined by extending pointwisely the standard operator. In case the domain $\mathbb{T}^\sharp$ is infinite, a widening operator can be defined for $D^\sharp_{\mathbb{P},\mathbb{M}}$ in the same way.

The resulting analysis is sound, in the sense of Theorem 4.3.3, since the abstract interpreter satisfies the assumption in Definition 4.3.8.

Partitioning, backward interpreter: The extension of the backward abstract interpreters defined in Section 3.1.2 and Section 3.3.2 is similar. The only difference is that the abstract transfer function for the **cnt**-statement should be based on a counterpart for the removal of directives from the top of tokens:

Definition 6.2.5. Abstract pop operation.

An abstract push operation *is a function* $\mathit{pop} : \mathbb{T}^\sharp \times \mathcal{D} \rightarrow \mathcal{P}(\mathbb{T}^\sharp)$*, such that:*

$$\forall t_0^\sharp \in \mathbb{T}^\sharp,\ \partial \in \mathcal{D},\ (\partial :: t) \in \gamma_{\mathbb{T}}(t_0^\sharp),\ \exists t_1^\sharp \in \mathit{pop}(t_0^\sharp, \partial),\ t \in \gamma_{\mathbb{T}}(t_1^\sharp)$$

This operator corresponds to the converse of push.

Implementation of the partitioning domain: We implemented a generic domain for this form of trace partitioning in ASTRÉE, i.e. a layer (below the control-based trace partitioning described in Chapter 5: an abstract value consists in a control-based partition of partitions based on the **cnt**-statements of abstract elements in $D^\sharp_{\mathbb{M}}$) which inputs the abstraction $\mathbb{T}^\sharp, \gamma_{\mathbb{T}}$ as a parameter. It currently works only for finite abstractions, which are specified by automata chosen by the user, as explained in Section 6.3.

Possible extension with dynamic partitioning: At the time of the writing, dynamic partitioning was not implemented, since it has not turned out necessary yet. In fact, the partitioning domain is largely related to the nature of the properties we wish to express, e.g. in semantic slicing .(introduced in Chapter 7), so that we do not expect the choice of the partitions to come out of the analysis.

However, the approach presented in this Section could be extended into a dynamic partitioning by:

- fixing a hierarchy of token abstractions for $P_{\mathbb{T}}$;
- choosing an initial abstraction (for instance, the trivial extension of P, i.e. the abstraction mapping any token $t \in \mathbb{T}$ into t_ϵ);
- defining an abstract semantics for the **cnt**-statement which may refine token abstraction, e.g. by creating new abstract tokens;
- implementing a widening operator, which should enforce the termination of the token abstraction refinement process.

In the following of this chapter, we provide several instantiations for the abstract domain $\mathbb{T}^\sharp$. Despite dynamic partitioning has not been implemented yet, we propose some domains, which may lead to powerful analyses, even though the design of adapted widening operators appears like a major issue.

6.3 Automata as Abstractions

6.3.1 Languages and automata

In this section, we propose to do a simple restriction: we assume that $\mathbb{T}$ is finite; hence, it is naturally equivalent to an automaton. As a consequence, the partitioning of the traces is based on the state reached in a finite automaton, by reading the word corresponding to the token defined in Definition 6.1.2.

Before we state the abstractions, we fix some notations. For a comprehensive introduction to automata, we refer the reader to [Knu62].

We write $\mathbb{Q}$ for a set of states; an automaton defines a transition relation over a subset of $\mathbb{Q}$, indexed with directives in $\mathcal{D}$:

Definition 6.3.1. Automaton.

An automaton $\mathcal{A}$ *is a triple* $(\mathbb{Q}_\mathcal{A}, q^{\mathrm{i}}_\mathcal{A}, \rightsquigarrow_\mathcal{A})$*, where:*

- $\mathbb{Q}_\mathcal{A} \subseteq \mathbb{Q}$ *is a* finite *set of* states*;*
- $q^{\mathrm{i}}_\mathcal{A}$ *is the* initial state*;*
- $(\rightsquigarrow_\mathcal{A}) \subseteq \mathbb{Q}_\mathcal{A} \times \mathcal{D} \times \mathbb{Q}_\mathcal{A}$ *is the* transition relation*.*

  In case $(q, \partial, q') \in (\rightsquigarrow_\mathcal{A})$*, we also write* $q \overset{\partial}{\rightsquigarrow}_\mathcal{A} q'$*.*

Moreover, we use the standard graphical representation for automata. We write $\mathbb{A}$ for the set of finite automata over the set of directives.

Definition 6.3.2. Semantics of an automaton.

Let $\mathcal{A}$ be an automaton $(\mathbb{Q}_{\mathcal{A}}, q^{\mathrm{i}}_{\mathcal{A}}, \rightsquigarrow_{\mathcal{A}})$. Each state q of $\mathcal{A}$ recognizes a language $\mathcal{L}[q] \subseteq \mathbb{T}$, defined by induction by the following rules (this definition is equivalent to a least fixpoint definition):

- $\epsilon \in \mathcal{L}[q^{\mathrm{i}}_{\mathcal{A}}]$;
- *if* $t \in \mathcal{L}[q]$ *and* $q \overset{\partial}{\rightsquigarrow}_{\mathcal{A}} q'$*, then* $(\partial :: t) \in \mathcal{L}[q']$.

We say that an automaton is *adequate* if it satisfies the following property:

$$\forall q \in \mathbb{Q}_{\mathcal{A}},\ \partial \in \mathcal{D},\ \exists q' \in \mathbb{Q}_{\mathcal{A}},\ q \overset{\partial}{\rightsquigarrow}_{\mathcal{A}} q'$$

Intuitively, an adequate automaton should have "enough transitions" so that there is no blocking configuration; as a consequence, any concrete token can be represented: $\forall t \in \mathbb{T},\ \exists q \in \mathbb{Q}_{\mathcal{A}},\ t \in \mathcal{L}[q]$.

6.3.2 Abstraction based on automata

The abstraction: Clearly, an adequate finite automaton provides exactly the structure required for a finite token abstraction (Section 6.2.1) and an abstract interpreter (Section 6.2.2) to be defined, as stated in the following theorem, which also defines the *automata-based abstraction*:

Theorem 6.3.1. Automata-based abstraction.

Let $\mathcal{A}$ be an adequate automaton. Then, the following set-up defines a valid token abstraction:

- $\mathbb{T}^\sharp = \mathbb{Q}_{\mathcal{A}}$;
- $\gamma_{\mathbb{T}} : (q \in \mathbb{Q}_{\mathcal{A}}) \mapsto \mathcal{L}[q]$;
- $t^\sharp_\epsilon = q^{\mathrm{i}}_{\mathcal{A}}$;
- $\mathit{push} : ((q, \partial) \in \mathbb{Q}_{\mathcal{A}} \times \mathcal{D}) \mapsto \{q' \in \mathbb{Q}_{\mathcal{A}} \mid q \overset{\partial}{\rightsquigarrow}_{\mathcal{A}} q'\}$.

We write $P_{\ll \mathcal{A} \gg}$ for the abstract extended system resulting from the application of the abstraction defined by $\mathcal{A}$ to the extended system $P_{\mathbb{T}}$.

Proof.

The above elements straightforwardly define an abstract extended system in the sense of Definition 6.2.2. □

Issues about a dynamic partitioning domain: The extension of this family of static abstractions into a dynamic partitioning abstraction would require a widening operator to be defined on the set $\mathbb{A}$ of finite automata and also a dynamic process to refine the structure during the analysis.

One the most promising approaches to the first problem consists in tree schemata [Mau00]. Tree schemata are designed so as to represent possibly infinite sets of trees; moreover, they and can be extended with counters [Mau99, §5], which could be used as the basis for defining widening operators on abstractions for sets of trees.

6.3.3 Examples

We now give a few examples of abstractions based on automata.

Control-based partitioning: Some cases of control-based partitioning described in Section 5.2.1 can be handled with the partitioning based on automata approach, as shown in the two following examples:

Definition 6.3.1. Loop unrolling.
Let us consider the program below:

$$\begin{array}{ll}
\ell_0: & \mathbf{while}(b)\{ \\
\ell_1: & \quad \mathbf{cnt}; \\
\ell_2: & \quad \ldots \\
\ell_3: & \}
\end{array}$$

Then, the abstraction defined by the automaton below allows to perform a loop unrolling of the first two iterations:

$$\begin{tikzcd}
q_0 \arrow[r, "\partial_{\ell_1}"] & q_1 \arrow[r, "\partial_{\ell_1}"] & q_2 \arrow[loop below, "\partial_{\ell_1}"]
\end{tikzcd}$$

Indeed, q_0 stands for the traces which did not enter the loop; q_1 stands for the traces which entered the loop body exactly once; q_2 stands for the traces which went through point ℓ_1 at least twice (i.e., after two or more iterations in the loop). Therefore, the abstraction based on this automaton is adapted to the partitioning of the first iteration of the loop in the program.

Definition 6.3.2. Conditional partitioning.
Let us consider the program below:

$$\begin{array}{ll}
\ell_0: & s_0; \\
\ell_1: & \mathbf{if}(c)\{ \\
\ell_2: & \quad \mathbf{cnt}; \\
\ell_3: & \quad s_1 \\
 & \}\mathbf{else}\{ \\
\ell_4: & \quad \mathbf{cnt}; \\
\ell_5: & \quad s_2\} \\
\ell_6: & s_3;
\end{array}$$

Then, the partitioning of the traces by the branch of the **if***-statement they went through can be simulated by a partitioning based on the automaton below:*

$$\begin{tikzcd}
 & q_0 \arrow[dl, "\partial_{\ell_2}"'] \arrow[dr, "\partial_{\ell_3}"] & \\
q_1 & & q_2
\end{tikzcd}$$

Indeed, q_1 (resp. q_2) should collect the traces which entered into the **true** *(resp.* **false***) branch of the conditional.*

However, the abstraction presented in Section 6.3.2 does not implement the dynamic partitioning strategies, which we designed in Section 5.2; in particular it is not adapted to the analysis of value-based partitioning. Moreover, the data-structures described in Section 5.3 allow for more efficient algorithms.

Discriminating traces: Therefore, we propose now a series of abstractions, which allow to discriminate sets of traces achieving various properties. In this paragraph, we elaborate on the theme of the simple loop of Example 6.3.1:

ℓ_0 : **while**$(b)\{$
ℓ_1 :   **cnt**;
ℓ_2 : $\ldots$
ℓ_3 : $\}$

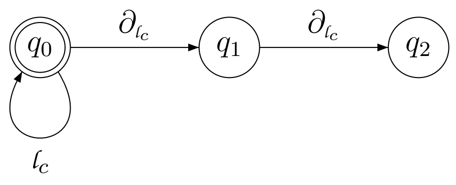
Definition 6.3.3. Iterations parity.

The automaton below allows to partition the traces with the parity of the number of iterations in the loop as a criterion:

$$\begin{tikzcd}
q_e \arrow[r, bend left, "\partial_{\ell_1}"] & q_o \arrow[l, bend left, "\partial_{\ell_1}"]
\end{tikzcd}$$

Clearly, q_e (resp. q_o) abstracts the executions which went through ℓ_1 an even (resp. odd) number of times. This automaton is adequate to analyze the program displayed in Figure 4.1(b).

Definition 6.3.4. Last iterations.
The automaton below allows to partition the traces so that the last two iterations are distinguished.



This is particularly useful in order to analyze the behavior of a program under the assumption that some even occurs in the last iteration, and to infer that some other property holds in the previous iteration(s). Semantic slicing (Chapter 7) will exploit this kind of abstractions, so as to characterize the states in the last iterations before some event occurs (e.g., an error).

More complex properties: Now, we come back to the example with two counter statements, which was presented in Example 6.1.1. More precisely, we envisage a program derived from the code in Example 6.1.1 and attempt to prove some property about it.

First, we formalize the abstraction introduced in Example 6.2.1:

Definition 6.3.5. Back to Example 6.2.1.
We let $\mathcal{A}$ be the automaton depicted in Figure 6.1(a). Then, this automaton defines the

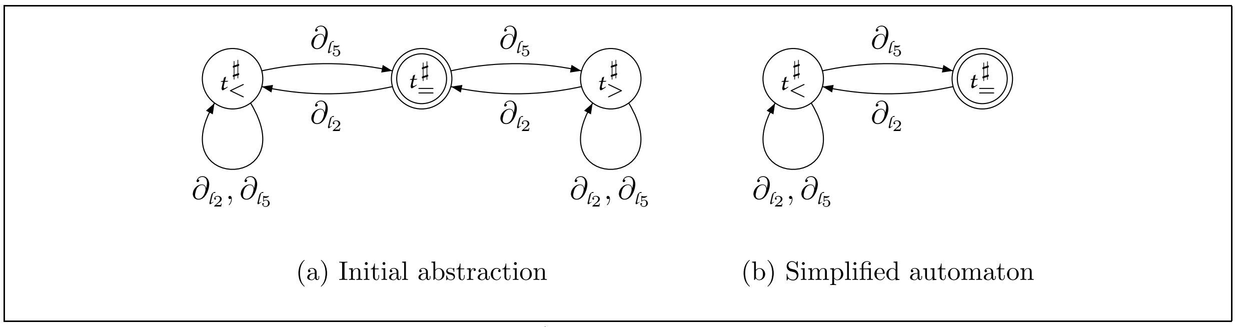


Figure 6.1: Abstractions as automata

same abstraction as we described in Example 6.2.1 and Example 6.2.2.
Basically, a simple reachability analysis would prove that the state $t^\sharp_>$ is useless: the statement ℓ_2 : **cnt** *is executed at least as often as ℓ_5 :* **cnt**. *Therefore, we could use a more simple automaton as well, with only two states $t^\sharp_<, t^\sharp_=$, despite it is not adequate; this automaton is displayed in Figure 6.1(b) (it can be used in the analysis since the token $t^\sharp_>$ simply is not generated during the analysis).*

A simple program, with the same structure is displayed in Figure 6.2. This program contains two counters (which we assume to have natural integer values): i is incremented whenever ℓ_2 : **cnt** is executed; j is incremented whenever ℓ_5 : **cnt** is executed (in the beginning, both counters are equal to 0). Our purpose is to provide an instantiation automaton to the generic partitioning abstract interpreter defined in Section 6.2.2, so as to prove the property:

$$i = j \text{ at point } \ell_1 \Longrightarrow b \text{ is always } \mathbf{true}$$

```
ℓ0 : bool b;
     int i, j = 0;
ℓ1 : while(true){
ℓ2 :    cnt;
        i = i + 1;
ℓ3 :    input(b);
ℓ4 :    if(b){
ℓ5 :       cnt;
           j = j + 1;
ℓ6 :    }
ℓ7 : }
ℓ8 : ...
```

(a) Program

$$\left\{\begin{array}{l} i = j \text{ at point } \ell_1 \\ \qquad \Longrightarrow b \text{ is always } \mathbf{true} \end{array}\right.$$

or equivalently:

$$\left\{\begin{array}{l} i = j \text{ at point } \ell_1 \\ \qquad \Longrightarrow \partial_{\ell_5} = \partial_{\ell_2} \end{array}\right.$$

(b) Property

$$\begin{tikzcd} t^\sharp_< \arrow[loop below, "{\partial_{\ell_2}, \partial_{\ell_5}}"] \arrow[r, bend left, "\partial_{\ell_5}"] & t^\sharp_= \arrow[l, bend left, "\partial_{\ell_2}"] \end{tikzcd}$$

(c) Initial abstraction

$$\begin{tikzcd} t^\sharp_{<-1} \arrow[loop below, "{\partial_{\ell_2}, \partial_{\ell_5}}"] \arrow[r, bend left, "\partial_{\ell_5}"] & t^\sharp_{=-1} \arrow[l, bend left, "\partial_{\ell_2}"] \arrow[r, bend left, "\partial_{\ell_5}"] & t^\sharp_= \arrow[l, bend left, "\partial_{\ell_2}"] \end{tikzcd}$$

(d) Successful abstraction

Figure 6.2: Two counters

Definition 6.3.6. Failed partitioning analysis.

We first attempt to prove it using the automaton proposed in Example 6.3.5, after *simplification (Figure 6.2(c)).*
We assume that the analysis resorts to the octagon abstract domain [Min01]; in fact we mostly care about the range for $i - j$. We assume that the analyzer uses the trivial iteration strategy, i.e. it computes a local invariant for ℓ_{i+1} after an invariant is found for ℓ_i except for the loop: after an invariant is found for ℓ_7, it re-computes an invariant for ℓ_1. Stabilization should be observed at the loop head ℓ_1. Last we assume that the

first iterations use the $\sqcup$ operator (so that delaying widening would not improve the final invariant).
The table below displays the most significant first steps in the analysis (note that local invariants are functions from tokens into numerical invariants):

Point	*Invariant*	
l_1	$\begin{cases} t^\sharp_= \mapsto i-j=0 \\ t^\sharp_< \mapsto \bot \end{cases}$	*initialization*
l_3	$\begin{cases} t^\sharp_= \mapsto \bot \\ t^\sharp_< \mapsto i-j=1 \end{cases}$	*first iteration in the loop*
l_6	$\begin{cases} t^\sharp_= \mapsto i-j=0 \\ t^\sharp_< \mapsto i-j=1 \end{cases}$	
l_7	$\begin{cases} t^\sharp_= \mapsto i-j=0 \\ t^\sharp_< \mapsto i-j=1 \end{cases}$	
l_1	$\begin{cases} t^\sharp_= \mapsto i-j=0 \\ t^\sharp_< \mapsto i-j\in[0,1] \end{cases}$	*abstract join*
l_3	$\begin{cases} t^\sharp_= \mapsto \bot \\ t^\sharp_< \mapsto i-j=1 \end{cases}$	*second iteration in the loop*
l_6	$\begin{cases} t^\sharp_= \mapsto i-j\in[0,1] \\ t^\sharp_< \mapsto i-j\in[0,1] \end{cases}$	
l_7	$\begin{cases} t^\sharp_= \mapsto i-j\in[0,1] \\ t^\sharp_< \mapsto i-j\in[0,2] \end{cases}$	
l_1	$\begin{cases} t^\sharp_= \mapsto i-j\in[0,1] \\ t^\sharp_< \mapsto i-j\in[0,2] \end{cases}$	*union,* property lost
l_1	$\begin{cases} t^\sharp_= \mapsto i-j\geq 0 \\ t^\sharp_< \mapsto i-j\geq 0 \end{cases}$	*widening, and stabilization*
l_3	$\begin{cases} t^\sharp_= \mapsto \bot \\ t^\sharp_< \mapsto i-j\geq 1 \end{cases}$	*stable, final invariant*
l_6	$\begin{cases} t^\sharp_= \mapsto i-j\geq 0 \\ t^\sharp_< \mapsto i-j\geq 0 \end{cases}$	*stable, final invariant*
l_7	$\begin{cases} t^\sharp_= \mapsto i-j\geq 0 \\ t^\sharp_< \mapsto i-j\geq 0 \end{cases}$	*stable, final invariant*

Clearly, the analysis fails to prove the property of interest, from the beginning of the third iteration in the loop. The reason for this failure is that all traces of the program ending in point l_3 are in the partition corresponding to $t^\sharp_<$; from this point, the analysis does not distinguish a trace such that b is always true (i.e., which always went through the true branch of the conditional) and a trace such that b is not always true (e.g., not in the first iteration).
Therefore, another abstraction should be considered.

Definition 6.3.7. Successful partitioning analysis.

We propose a new abstraction so as to distinguish the traces such that b has always been true but *the conditional has not been ran in the last iterations and the traces which do not enjoy that property. The corresponding automaton is depicted on Figure 6.2(d); note that the state $t^\sharp_<$ is split into two states $t^\sharp_{=-1}$ and $t^\sharp_{<-1}$, with the following concretizations:*

$$\begin{array}{rrcl} \gamma_{\mathbb{T}} : & t^\sharp_{=-1} & \mapsto & \{t \mid \mathbf{occurences}(\!|\partial_{l_5} \in t|\!) = \mathbf{occurences}(\!|\partial_{l_2} \in t|\!) - 1\} \\ & t^\sharp_{<-1} & \mapsto & \{t \mid \mathbf{occurences}(\!|\partial_{l_5} \in t|\!) < \mathbf{occurences}(\!|\partial_{l_2} \in t|\!) - 1\} \end{array}$$

We sketch the analysis in the table below:

Point	*Invariant*	
ℓ_1	$\begin{cases} t^\sharp_= & \mapsto & i-j=0 \\ t^\sharp_{=-1} & \mapsto & \bot \\ t^\sharp_{<-1} & \mapsto & \bot \end{cases}$	*initialization*
ℓ_3	$\begin{cases} t^\sharp_= & \mapsto & \bot \\ t^\sharp_{=-1} & \mapsto & i-j=1 \\ t^\sharp_{<-1} & \mapsto & \bot \end{cases}$	*first iteration in the loop*
ℓ_6	$\begin{cases} t^\sharp_= & \mapsto & i-j=0 \\ t^\sharp_{=-1} & \mapsto & \bot \\ t^\sharp_{<-1} & \mapsto & \bot \end{cases}$	
ℓ_7	$\begin{cases} t^\sharp_= & \mapsto & i-j=0 \\ t^\sharp_{=-1} & \mapsto & i-j=1 \\ t^\sharp_{<-1} & \mapsto & \bot \end{cases}$	
ℓ_1	$\begin{cases} t^\sharp_= & \mapsto & i-j=0 \\ t^\sharp_{=-1} & \mapsto & i-j=1 \\ t^\sharp_{<-1} & \mapsto & \bot \end{cases}$	*union, second abstract iteration*
ℓ_3	$\begin{cases} t^\sharp_= & \mapsto & \bot \\ t^\sharp_{=-1} & \mapsto & i-j=1 \\ t^\sharp_{<-1} & \mapsto & i-j=2 \end{cases}$	
ℓ_6	$\begin{cases} t^\sharp_= & \mapsto & i-j=0 \\ t^\sharp_{=-1} & \mapsto & i-j=1 \\ t^\sharp_{<-1} & \mapsto & i-j=1 \end{cases}$	
ℓ_7	$\begin{cases} t^\sharp_= & \mapsto & i-j=0 \\ t^\sharp_{=-1} & \mapsto & i-j=1 \\ t^\sharp_{<-1} & \mapsto & i-j\in[1,2] \end{cases}$	*end of the second iteration*
ℓ_1	$\begin{cases} t^\sharp_= & \mapsto & i-j=0 \\ t^\sharp_{=-1} & \mapsto & i-j=1 \\ t^\sharp_{<-1} & \mapsto & i-j\in[1,2] \end{cases}$	*union, beginning of the third iteration*
$\ldots$	$\ldots$	
ℓ_1	$\begin{cases} t^\sharp_= & \mapsto & i-j=0 \\ t^\sharp_{=-1} & \mapsto & i-j\in[1,2] \\ t^\sharp_{<-1} & \mapsto & i-j\in[1,3] \end{cases}$	*union, beginning of the fourth iteration*

The invariant would stabilize if we apply widening right after this point. After stabilization, we get the following invariant:

ℓ_1	$\begin{cases} t^\sharp_{=} & \mapsto \; i - j = 0 \\ t^\sharp_{=-1} & \mapsto \; i - j \geq 1 \\ t^\sharp_{<-1} & \mapsto \; i - j \geq 1 \end{cases}$	*loop invariant*
ℓ_3	$\begin{cases} t^\sharp_{=} & \mapsto \; \bot \\ t^\sharp_{=-1} & \mapsto \; i - j = 1 \\ t^\sharp_{<-1} & \mapsto \; i - j \geq 2 \end{cases}$	
ℓ_6	$\begin{cases} t^\sharp_{=} & \mapsto \; i - j = 0 \\ t^\sharp_{=-1} & \mapsto \; i - j \geq 1 \\ t^\sharp_{<-1} & \mapsto \; i - j \geq 1 \end{cases}$	
ℓ_7	$\begin{cases} t^\sharp_{=} & \mapsto \; i - j = 0 \\ t^\sharp_{=-1} & \mapsto \; i - j \geq 1 \\ t^\sharp_{<-1} & \mapsto \; i - j \geq 1 \end{cases}$	

Obviously, the property of interest is proved, since at ℓ_1, for $t^\sharp_{=}$, $i = j$.

The above example shows how the trace partitioning proposed in this chapter can be helpful in proving user properties. The same kind of technique will also be most useful in the case of semantic slicing, introduced in Chapter 7.

6.4 Numeric Abstractions

6.4.1 Parikh abstraction

We propose a second family of abstractions, which is based on the number of occurrences of each directive in a token. More precisely, if we let p denote the number of **cnt**-statements in the program, each abstraction is defined by a numeric abstractions for sets of vectors of $\mathbb{N}^p$, where a vector collects the number of times each directive was encountered.

The first step consists in the Parikh vector [Par66] abstraction:

Definition 6.4.1. Parikh abstraction.

We let $\mathbb{T}^\sharp{}_{\mathfrak{P}} = \mathcal{P}(\mathcal{D} \rightarrow \mathbb{N})$ *be the* Parikh abstraction domain*, with the pointwise ordering. The abstraction function* $\gamma_{\mathbb{T}_{\mathfrak{P}}} : \mathbb{T}^\sharp{}_{\mathfrak{P}} \rightarrow \mathcal{P}(\mathbb{T})$ *is defined by:*

$$\gamma_{\mathbb{T}_{\mathfrak{P}}} : \quad \Phi \quad \mapsto \quad \{\partial_{\ell_0} :: \ldots :: \partial_{\ell_n} \mid \exists \phi \in \Phi,\ \forall \partial \in \mathcal{D},\ \mathbf{Card}(()\{i \in \mathbb{N} \mid \partial_{\ell_i} = \partial) = \phi(\partial)\}$$

Moreover, we define:

- *the abstract initial token* $t^\sharp_{\epsilon\mathfrak{P}} = \{\lambda(\partial \in \mathcal{D}) \cdot 0$*;*
- *the abstract push operation* $\mathit{push}_{\mathfrak{P}} : \mathbb{T}^\sharp{}_{\mathfrak{P}} \times \mathcal{D} \rightarrow \mathbb{T}^\sharp{}_{\mathfrak{P}}$ *defined by:*

$$\begin{array}{rlcl} \mathit{push}_{\mathfrak{P}} : & (\{\phi\}, \partial) & \mapsto & \left\{ \lambda(\partial' \in \mathcal{D}) \cdot \left\{ \begin{array}{ll} \phi(\partial) + 1 & \mathit{if}\ \partial = \partial' \\ \phi(\partial') & \mathit{otherwise} \end{array} \right. \right\} \\ & (\Phi, \partial) & \mapsto & \{\mathit{push}_{\mathfrak{P}}(\phi, \partial) \mid \phi \in \Phi\} \end{array}$$

The Parikh abstraction maps any token ι into a function which associates to any directive ∂ the number of times it appears in ι. The initial token $\iota^\sharp_{\epsilon\mathfrak{P}}$ maps any directive to 0 (the abstract initial token contains no directive). The abstract push operation increments by one the image of the token pushed in the Parikh abstraction (which corresponds to the action of a concrete token push).

This set up straightforwardly defines an abstract extended system in the sense of Definition 6.2.2.

6.4.2 Composing numerical abstractions

Definition: The second steps consists in applying a numerical abstraction to the sets of vectors (Definition 6.4.1).

Definition 6.4.2. Vector abstraction.

Let $D^\sharp_{\mathbb{M}}$ be a numeric domain, for representing sets of functions from ∂ to integer values.

Such an abstraction trivially composes with the Parikh abstraction. In particular,
- the initial token should be an abstract value approximating $\iota^\sharp_{\epsilon\mathfrak{P}}$;
- the abstract push operation derives from the common *assign* operator.

Examples of abstractions: Various numerical abstractions can be used as a vector abstraction:
- *k-Limiting*, which amounts to replacing any value larger than some integer k with $\top$ in the Parikh vectors (so that we ge a finite domain);
- *Congruences*, so as to express, e.g., cyclic behaviors, properties on counters, cyclic buffers (like buffers stored in arrays).
- *Affine equalities* (Karr), so as to express that the number of times two events happened are equal up to some constant;
- *Difference bound matrices*, so as to express that the number of times event e_1 happened is smaller than the number of times event e_2 happened plus some constant;

At the time of the writing of this thesis, we found only k-limiting and congruences abstraction useful. These abstractions were used in semantic slicing (Chapter 7).

Remark 6.4.1. Widening operators.

If we use an infinite numeric domain to abstract Parikh vectors, a widening operator is needed so as to ensure the convergence of the iteration (Definition 4.3.6). Note that this operator apply to partitions, i.e. to elements of $\mathcal{P}(D^\sharp_{\mathbb{M}})$. As a consequence, the definition of widening operators for such domains is a non-trivial issue.
At this time, we have not implemented such a domain yet, so this is a major area for future work.

Comparison with the "automaton-based abstraction": We proposed two families of domains. The first one involves automata, and is adapted for the case of finite, known in advance abstractions. In particular, it allows to express properties about the order the events occur in. On the other hand, the extension into a dynamic partitioning seems a rather tedious issue, which should require completely new domains to be defined.

The second one is purely based on numerical abstractions; it forgets everything about the order the event occur in. A finite numerical abstraction may be reduced into an automaton as well; however, the advantage with the numerical approach is that only the tokens which are needed are created at analysis time, since this approach creates abstract tokens dynamically (whereas the automata should be completely defined before the analysis starts).

Overall, both families of domains are rather complementary, so that it seems interesting to implement an equivalent for the reduced product (Definition 3.1.1), in the case of token abstractions.

Part III

Alarm Inspection and Semantic Slicing

Chapter 7

Semantic Slicing

Static analyzers like ASTRÉE [BCC$^+$02, BCC$^+$03a] are *sound* but *incomplete*: the results of an analysis are provably sound, but the analysis mail fail to establish true properties. The alarms produced by a static analyzer are a major issue for end-users, since an alarm may result either from a true error or from an imprecision of the analysis.

We propose to extract *semantic slices*, i.e. to characterize a subset of the trace semantics of a program with abstract invariants, so as to provide a better view of the concrete context of an alarm raised by a static analyzer. Semantic slices can be used either to prove an alarm false or to design and check real error scenarios.

We proposed this framework in [Riv05b].

We detail the motivations for semantic slicing in Section 7.1. We describe *semantic slicing criteria* in Section 7.2. Then, we provide algorithms for the extraction of semantic slices in Section 7.3. We conclude in Section 7.4 with examples, early results in the implementation of our technique in the ASTRÉE analyzer and comparisons with other techniques.

7.1 Why to Extract Semantic Slices ?

7.1.1 Incompleteness of static analysis: alarms and errors

In this chapter, we consider static analyses that aim at proving the absence of runtime errors such as ASTRÉE ; yet, our algorithms could apply to other safety analyses as well. The most favorable result of a static analysis is the success of the proof that the analyzed program is safe, i.e. that it causes no a runtime error whatever the inputs. However, buggy programs exist, so one may expect errors to be found by static analyzers.

However, static analyzers usually are *not complete*: analyzers like ASTRÉE cannot compute the most precise invariant for any program, due to the imprecision inherent in the abstraction, in the abstract operations, in the join and in the widening. In particular, they may fail to prove a program safe, despite the fact that it is not dangerous, due to excessively imprecise invariants. In this case, the analyzer will also report an alarm.

As a consequence, alarms are a major issue for end-users. Indeed, an alarm reported by the analyzer means that the program *may* be unsafe but *does not prove* that it is indeed unsafe; therefore, the user needs to tell the “true” alarms from the “false” alarms, in order to decide whether to fix the program or to consider it safe. Figure 7.1 presents three examples of programs causing an analyzer to raise an alarm.

- Let us consider the code in Figure 7.1(a); in particular, we focus on the assertion in the end of the program. This program is safe, since $|x| > 10 \Rightarrow (x < -10 \vee 10 < x)$. Not all analyzers would infer this property. Indeed, proving the safety of the assertion at l_8 requires carrying out a relation among boolean and integer variables. For instance, ASTRÉE handles such relations, but may not infer any relation between b, x, y, in case the relation packing strategy mentioned in Section 5.1.3 chooses not to include these variables in a same *pack*; in this case, the assertion at l_8 would not be proved correct and the analyzer would raise an alarm. A very simple tuning of the packing strategy would solve the alarm; yet, a user may need some help from the analyzer before proposing the right hint, especially if non specialist in static analysis. We would also expect a refining analysis to help proving the safety of this program.
- The program displayed in Figure 7.1(b) is not safe. Indeed, if the input at l_4 is negative, then the assertion at l_5 fails in the next iteration. Obviously, an analyzer like ASTRÉE would report an alarm in such a case; yet, a non-experienced user may need some more precise information in order to understand the problem and fix the program. In particular, an error scenario would be particularly helpful in order to understand the origin of the failure.
- For the sake of the example, we assume that machine integer values are mathematical integers (no overflows). Then, the program presented in Figure 7.1(c) is safe, since it is well known that $\forall x, y, z \in \mathbb{Z},\ \forall n \geq 3,\ x^n = y^n + z^n \Longrightarrow x = y = z = 0$ (Fermat’s theorem, proved in [Wil95]). However, the proof for this property is far beyond the abilities of any static analyzer at the time we write this thesis: the proof basically required more than 300 years of research since the theorem was stated for the first time; and it does not seem feasible to automatize such a process (note that there exist simpler proofs for small values of n, but proving the program safe would require proving the property for *any* integer n, so it amounts to proving Fermat’s theorem). Therefore, any analyzer like ASTRÉE would fail on this example, because the abstract domain would be limited to a set of properties and logical formulas which does not allow expressing a proof of the mathematical theorem involved. As a consequence, we do not intend to provide a verification of *any* safe program.

The above series of examples shows two cases we intend to improve the analysis of: in the case of Figure 7.1(a), we expect to refine an analysis and show the safety of the program; in the case of Figure 7.1(b), we expect an example of error; finally, the case of Figure 7.1(c) is particularly involved and is not addressed in this thesis.

```
ℓ0 : float x, y; bool b;
ℓ1 : input(x);
ℓ2 : if(x > 0){
ℓ3 :     y = x;
ℓ4 : }else{
ℓ5 :     y = −x;
ℓ6 : }
ℓ7 : b = (y > 10);
ℓ8 : assert(b ⇒ (x < −10 ∨ 10 < x));
```

(a) False alarm (absolute value)

```
ℓ0 : int x, y;
ℓ1 : x = 1;
ℓ2 : y = 1;
ℓ3 : while(true){
ℓ4 :     input(x);
ℓ5 :     assert(y > 0);
ℓ6 :     y = x;
ℓ7 : }
```

(b) True error

```
ℓ0 : int x, y, z, n;
ℓ1 : input(x);
ℓ2 : input(y);
ℓ3 : input(z);
ℓ4 : input(n);
ℓ4 : if(n ≥ 3 ∧ x > 0
           ∧y > 0 ∧ z > 0){
ℓ5 :     assert(x^n ≠ y^n + z^n);
ℓ6 : }
```

We assume machine integers are not bounded (no modular arithmetics; $\mathbb{I} = \mathbb{Z}$)

Mathematical property:

$$\forall(x, y, z) \in \mathbb{Z}^3,\ (\exists n \in \mathbb{N},\ n \geq 3 \land x^n = y^n + z^n) \Longrightarrow x = 0 \lor y = 0 \lor z = 0$$

(c) Verification requiring a very complicated theorem

Figure 7.1: Cases of alarms

7.1.2 Semantic slices

Our goal is to provide some support in the alarm investigation process. We propose resorting to automatic, sound static analysis techniques so as to refine an initial static analysis into an approximation of a subset of traces that actually lead to an error (aka, set of *erroneous traces*).

In particular, if we consider the case of a safe program, such as the piece of code presented in Figure 7.1(a), the set $\mathcal{E}$ of erroneous traces is empty. The alarm follows from the failure of the analyzer to prove the emptiness of $\mathcal{E}$. In case a refinement of the initial analysis proves that $\mathcal{E} = \emptyset$, then the program is proved safe by the refining analysis despite the failure of the initial analysis. We propose to perform this refinement by taking the error condition into account. Therefore, the refining analysis should include some *backward* phases.

In the case of a dangerous program, such as the fragment presented in Figure 7.1(b), the set of erroneous traces $\mathcal{E}$ is definitely not empty. Moreover, we wish to extract an error scenario, i.e. a set of conditions on the execution of the program, which entails the occurrence of an error at the alarm point. Therefore, we wish to exhibit a *witness*, i.e. a set of erroneous traces $\mathcal{E}' \subseteq \mathcal{E}$, such that $\mathcal{E}' \neq \emptyset$.

In the following, a subset of the traces of the program is called a *semantic slice*. The purpose of this chapter is to extract relevant semantic slices.

7.1.3 Extraction of semantic slices

A semantic slice is defined by a criterion, which combines a collection of constraints on program executions. Among the criterion we are going to consider, we can cite:

- **initial and final states,** so as to restrict to e.g., traces leading to some dangerous state(s);
- **execution patterns,** so as to restrict to some sets of paths in the control flow: for instance, we may choose to focus on the traces which iterate a loop at least twice, or on the traces which iterate a loop an even number of times;
- **input constraints,** so as to fix a set of inputs and to restrict to the traces corresponding to these inputs.

Semantic slicing criteria are abstractions for sets of traces. We describe precisely the various semantic slicing criteria in Section 7.2.

As usual, we wish to compute approximations for semantic slices. Therefore, we resort to the same approximations for sets of traces, derived from an approximation for sets of stores, as in Section 3.1.1. The extraction of a semantic slice will be based on a sequence of forward and backward analyses, which refine more and more the invariants. Static analyses for approximating semantic slices are described in Section 7.3.

Last, the ultimate goal would be to synthesize accurate semantic slicing criteria automatically, so as to propose a helpful scenario for an alarm. At the time we write this thesis, this point is still work in progress. Yet, an important tool for that is presented in the Chapter 8: abstract dependences aim at discovering chains of dependences among variables, which may cause an error to occur.

7.2 Semantic Slicing Criteria

7.2.1 Criteria as Abstractions

In this chapter, we consider a program P, defined by a tuple $(\mathbb{L}, \mathbb{X}, \rightarrow, \mathbb{S}^{\mathrm{i}})$ and its semantics $[\![P]\!] \subseteq \Sigma$, which we introduced in Definition 2.2.1.

A criteria for semantic slicing aims at defining a set of traces. Therefore, we define a criterion as an abstraction for a set of traces, and let a set of criteria be an abstract domain for representing sets of traces.

Definition 7.2.1. Semantic slicing criterion.

A semantic slicing domain *is an abstraction of sets of traces defined by a domain* $\mathbb{C}$ *and a concretization function* $\gamma_{\mathbb{C}} : \mathbb{C} \rightarrow \mathcal{P}(\Sigma)$.

The ordering $\sqsubseteq$ *of* $\mathbb{C}$ *is inherited from* $\gamma_{\mathbb{C}}$ *and the inclusion ordering over* $\mathcal{P}(\Sigma)$*:*

$$\forall c_0, c_1 \in \mathbb{C},\ c_0 \sqsubseteq c_1 \iff \gamma_{\mathbb{C}}(c_0) \subseteq \gamma_{\mathbb{C}}(c_1)$$

remove We call an element $c \in \mathbb{C}$ *a* semantic slicing criterion.

In practice, we will use semantic slicing in order to extract sets of traces of programs that do satisfy some conditions (described by the semantic slicing criterion); as a consequence, we define a semantic slice as the set of traces of a program, which also belong to the concretization of the criterion:

Definition 7.2.2. Semantic slice.

Let $\mathbb{C}$ *be a semantic slicing domain,* $c \in \mathbb{C}$*. Then, the* semantic slice *of the set of traces* $\mathcal{E}$ *(resp. of the program* P*) defined by the criterion* c *is the set of traces* $\mathfrak{Slice}_{\mathbb{C}}\langle \mathcal{E}, c\rangle$ *(resp.* $\mathfrak{Slice}_{\mathbb{C}}\langle [\![P]\!], c\rangle$*) defined by:*

$$\mathfrak{Slice}_{\mathbb{C}}\langle \mathcal{E}, c\rangle = [\![P]\!] \cap \gamma_{\mathbb{C}}(c)$$

(resp. $\mathfrak{Slice}_{\mathbb{C}}\langle [\![P]\!], c\rangle = [\![P]\!] \cap \gamma_{\mathbb{C}}(c)$*)*

At this point, all the abstractions of sets of traces presented before would work: abstraction with numerical invariants, with functions, with projections of traces... However, most of these abstractions would not be of the greatest interest here. Therefore, the following subsections review various useful semantic slicing criteria:

- initial and final states in Section 7.2.2;
- execution patterns in Section 7.2.3;
- constraints on the input values in Section 7.2.4.

7.2.2 Initial and Final States

The first semantic slicing criterion we consider is the restriction to a set of initial and final states. A slicing criterion consists in the data of a set of initial states and a set of final states; the concretization of such a criterion is the set of all traces from an initial state to a final state:

Definition 7.2.3. Final states slicing criterion.

The semantic slicing domain capturing "initial and final states" criteria is defined by:

- $\mathbb{C}_{\mathrm{i-f}} = \mathcal{P}(\mathbb{S}^{\mathrm{i}}) \times \mathcal{P}(\mathbb{S})$*;*
- $\gamma_{\mathrm{i-f}} : (\mathcal{I}, \mathcal{F}) \mapsto \{\langle s_0, s_1, \ldots, s_{n-1}, s_n\rangle \mid s_0 \in \mathcal{I},\ s_n \in \mathcal{F}\}$*.*

Note that we assume that the initial states specified in the criterion is a subset of the initial states of the program. We could remove this assumption and study slices defined by any set of initial states; however, such slices may not consist only in (parts of) real executions.

The most important application for such criteria consists in fixing a set of traces ending in a dangerous state. Then, deciding whether the program is unsafe amounts to checking that this slice is empty (the program is safe) or non empty (the program has an erroneous trace), as pointed out in Section 7.1.2.

In the following, we make the assumption that both $\mathcal{I}$ and $\mathcal{F}$ are of the form $\{\ell\} \times \mathscr{M}$, where $\mathscr{M} \subseteq \mathbb{M}$: the set of initial states of interest is defined by *one* control state and a set of memory states (and the same for the set of final states). This assumption is made so as to make the notations and technical developments more simple, even though it is also rather natural:

- a program has only *one* entry control state; $\mathcal{I}$ should just refine the initial condition on the executions;
- the purpose of $\mathcal{F}$ is to specify a final condition to investigate; it usually corresponds to an alarm raised by the static analyzer, so it usually also corresponds to only *one* control state.

A very efficient way to represent such criteria proceeds by choosing a control state ℓ, and an abstract invariant $d \in D^\sharp_{\mathbb{M}}$: indeed, such a pair defines a set of states $\{\ell\} \times \gamma_{\mathbb{M}}(d)$.

Definition 7.2.1. Semantic slicing based on the final state.
For instance, in the case of the program presented in Figure 7.1(a), we should study the semantic slice defined by the set of final states

$$\mathcal{F} = \{(\ell_8, \rho) \mid \rho \in \mathbb{M},\ \rho(b) \wedge -10 \leq \rho(x) \leq 10\}$$

This set of states can be represented as a pair (ℓ, d), as suggested above.

7.2.3 Execution Patterns

A second family of slicing criteria select sets of traces that satisfy some *control properties*, defined by abstractions introduced in Chapter 6. For instance, we may decide to focus on traces which spend an *even* number of iterations in a loop.

In this section, we assume that the user inserted some **cnt**-statements in the program, which correspond to special actions like "entering in a loop" or "running statement s". The criteria should account for sets of sequences of such actions (each action corresponds to the control state of the **cnt**-statement).

A criterion is defined by:

- an automaton $\mathcal{A} = (\mathbb{Q}_{\mathcal{A}}, q^{\mathrm{i}}_{\mathcal{A}}, \rightsquigarrow_{\mathcal{A}})$;
- a final states $q^{\mathrm{f}}_{\mathcal{A}} \in \mathbb{Q}_{\mathcal{A}}$, which recognizes the set of sequences of actions we want to extract.

We use the same notations as in Chapter 6: for instance, $\mathbb{T}$ denotes the set of sequences of actions; $P_{\mathbb{T}}$ stands for the extended system introduced in Section 6.1.3.

The criterion is defined formally as follows:

Definition 7.2.4. Execution patterns criterion.
The domain $\mathbb{D}_{\mathbb{A}}$ *of* execution patterns criteria *is defined by* $\mathbb{D}_{\mathbb{A}} = \mathbb{A} \times \mathbb{Q}$.
Let $(\mathcal{A}, q^{\mathrm{f}}_{\mathcal{A}}) \in \mathbb{D}_{\mathbb{A}}$*, where* $\mathcal{A} = (\mathbb{Q}_{\mathcal{A}}, q^{\mathrm{i}}_{\mathcal{A}}, \leadsto_{\mathcal{A}})$*. We write* $\tau : (\mathbb{Q}_{\mathcal{A}} \times \mathbb{T}) \to \mathbb{L}$ *for the standard erasure function. Then, we define the concretization* $\gamma_{\mathbb{A}}(\mathcal{A}, q^{\mathrm{f}}_{\mathcal{A}})$ *of* $(\mathcal{A}, q^{\mathrm{f}}_{\mathcal{A}})$ *by:*

$$\gamma_{\mathbb{A}}(\mathcal{A}, q^{\mathrm{f}}_{\mathcal{A}}) = \pi^{\Sigma}_{\tau}(\llbracket P_{\lessdot\mathcal{A}\gtrdot} \rrbracket^{\mathrm{P}}(q^{\mathrm{f}}_{\mathcal{A}}))$$

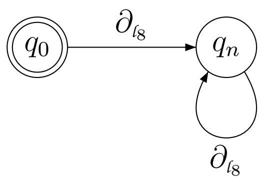
The above definition uses the abstract extended system to state the set of traces of P which also correspond to a path from $q^{\mathrm{i}}_{\mathcal{A}}$ to $q^{\mathrm{f}}_{\mathcal{A}}$ in $\mathcal{A}$ (the operator π^{Σ}_{τ} removes the states of $\mathcal{A}$ in the traces of $P_{\lessdot\mathcal{A}\gtrdot}$).

This family of criteria applies nicely to the distinction of loop iterations, as shown in the following example:

Definition 7.2.2. Criterion for the specification of execution patterns.
Let us consider the program in Figure 7.1(b). Obviously, it will not fail at the first iteration, since $y = 1$*; however, it may fail at any other iteration. Therefore, it would seem wise, to exclude the first iteration, when looking for a witness (i.e., erroneous trace).*
First, we add a **cnt***-statement anywhere inside the loop so as to count actions corresponding to some point in the loop body, i.e. the iteration number (Figure 7.2(a)).*
Second, we select an automaton, which allows to distinguish the "positive" iterations and a state in the automaton corresponding to this selection. For instance, we could choose the automaton $\mathcal{A}$ *displayed in Figure 7.2(b) and the state* q_n *(*q_0 *corresponds to the first iteration;* q_n *to any other iteration).*

```
ℓ0 :  int x, y;
ℓ1 :  x = 1;
ℓ2 :  y = 1;
ℓ3 :  while(true){
ℓ4 :     input(x);
ℓ5 :     assert(y > 0);
ℓ6 :     y = x;
ℓ8 :     cnt()
ℓ7 :  }
```

(a) Modified code



(b) Automaton

Figure 7.2: Exclusion of the first iteration

Similarly, we could slice the traces with an even number of iterations in a loop (as in Example 6.3.3) or the traces with at least 2 iterations and distinguish the last two iterations 6.3.4.

Remark 7.2.1. Precision improvement inherent in partitioning.

Here, the partitioning of the program guided by the choice of an automaton is targeted at the specification of a set of traces to extract, as a semantic slice. However, we shall note in the following sections that this choice may also improve the precision of the semantic slice, by helping the static analysis to infer more precise invariants, in a similar way as we did in Chapter 5.
For instance, distinguishing the last two iterations before some event occurs may help the backward analysis to produce better results, in the same way as forward analyses may benefit from the unrolling of the first iterations in a loop.

7.2.4 Input Constraints

A third family of slicing criteria discriminates traces characterized by the *values read by* **input***-statements*: a criterion defines a set of valid inputs for each **input**-statement; the semantic slice is the set of traces satisfying the property that all input values satisfy the criterion.

In fact, we can define a wider family of criterion, by enforcing constraints not only on the input values but also on any value, at any point in the program.

In the formal definition below, the criterion is represented with a function, which defines the set of valid input values.

Definition 7.2.5. Input constraints criterion.

The domain $\mathbb{C}_{\text{in}}$ of "input constraints criteria" is defined by $\mathbb{C}_{\text{in}} = (\mathbb{L} \times \mathbb{X}) \to \mathbb{V}$.
Let $\nu \in \mathbb{C}_{\text{in}}$. Then, the concretization of ν is defined by:

$$\gamma_{\text{in}}(\nu) = \{\langle(l_0, \rho_0), \ldots, (l_n, \rho_n)\rangle \in \Sigma \mid \forall i \in [\![0, n-1]\!],\ \forall x \in \mathbb{X},\ \rho_{i+1}(x) \in \nu(l_i, x)\}$$

Of course, in practice, only a few points and a few variables should be affected by the slicing, so that a sparse representation for function $\nu \in \mathbb{C}_{\text{in}}$ should be used.

Such a family of criteria is most useful in order to study the behavior of a program in presence of some special inputs, and also, in order to check that some set of inputs result in a crash of the program (i.e., to check an error scenario).

Definition 7.2.3. Input constraints and errors.

Let us consider again the program in Figure 7.1(b). We observed that an error occurs when the value of the input value for x at l_4 is negative. Indeed, if this value is negative, then at the next iteration, $y < 0$, so that the program crashes at l_5.
Therefore, we let ν be defined by:

$$\begin{array}{rlcl} \nu: & (\mathbb{L} \times \mathbb{X}) & \to & \mathbb{V} \\ & (l_4, x) & \mapsto & \{-1\} \\ & (l, v) \neq (l_4, x) & \mapsto & \mathbb{V} \end{array}$$

This criterion defines all the traces satisfying the condition that the input statement always reads the negative value -1.
However, not all these traces cause the program to fail. Indeed, a trace which does not complete more than one iteration in the loop does not end in an error state (we recall that the semantics we consider is prefix-closed).

Intuitively, we need to combine the above criterion with the criterion introduced in Example 7.2.2; this is the goal of the next subsection.

7.2.5 Combination of Criteria

Criteria can be combined thanks to a kind of product.

Definition 7.2.6. Product of criteria.

Let $(\mathbb{C}_0, \gamma_0)$ and $(\mathbb{C}_1, \gamma_1)$ be two semantic slicing criterion domains. We let the product semantic slicing criterion domain *$(\mathbb{C}_p, \gamma_p)$ be defined by:*

- $\mathbb{C}_p = \mathbb{C}_0 \times \mathbb{C}_1$;
- $\forall(c_0, c_1) \in \mathbb{C}_p,\ \gamma_p(c_0, c_1) = \gamma_0(c_0) \cap \gamma_1(c_1)$.

In particular, we can apply this construction to the semantic slicing criterion domains introduced in the previous subsections and combine the criteria introduced in Example 7.2.2 and Example 7.2.3:

Definition 7.2.4. Combination of semantic slicing criteria.

We consider the same program as in Example 7.2.2. Two semantic slicing criteria were introduced so as to study this example: the first one restricts to traces with more than one iteration in the loop; the second to traces characterized with negative inputs only.
The combination of both criteria following Definition 7.2.6 results in a set of traces which all crash *at point l_5. Moreover, the corresponding semantic slice is non-empty: clearly, this program has traces with more than one iterations and which always read negative values at l_4. As a consequence, this semantic slice defines a valid error scenario, which proves the program to be indeed buggy.*

7.3 Approximation of Slices Defined by Final States

7.3.1 Approximation of a Slice

We study the semantic slices defined by the data of a (set of) final state(s) (Section 7.2.2) first; the case of other semantic slicing criteria is the subject of the Section 7.4.

Principle of the abstraction: In this section, we consider a program P defined as usual and a pair of sets of states: $\mathcal{I}$ represents initial states; and $\mathcal{F}$ represents final states. We focus on the criterion $c = (\mathcal{I}, \mathcal{F}) \in \mathbb{C}_{\mathrm{i-f}}$, and wish to approximate the slice $\mathfrak{Slice}_{\mathbb{C}_{\mathrm{i-f}}}\langle [\![P]\!], c\rangle$.

The static analysis approximates the semantics of P with an invariant in the domain $D^\sharp = \mathbb{L} \rightarrow D^\sharp_{\mathbb{M}}$ (Section 3.1): it associates a local invariant to each control state l, which approximates the set of memory states observed at point l. Semantic slices should improve the understanding of the results of static analyses, and should be computed statically. As a consequence, we propose to define the semantic slice as an invariant in $D^\sharp$, which characterize a set of traces defined by the concretization of $D^\sharp$, introduced in Section 3.1.1. As a consequence, the domain for representing semantic slices takes an abstract domain for representing sets of stores as a parameter (in practice, we use the domain described in Section 5.1.3), which might be based on the many relational and non-relational abstractions available today (Section 3.1.3).

Definition 7.3.1. Approximation of semantic slices.

We keep the above notations. Formally, an approximation of the semantic slice $\mathfrak{Slice}_{\mathbb{C}_{\mathrm{i-f}}}\langle [\![P]\!], c\rangle$ *is an invariant* $\mathfrak{I} \in D^\sharp$*, such that:*

$$\mathfrak{Slice}_{\mathbb{C}_{\mathrm{i-f}}}\langle [\![P]\!], c\rangle \subseteq \gamma(\mathfrak{I})$$

In particular, semantic slicing strongly differs from regular slicing methods. In syntactic slicing [Wei81, HRB90], a criterion defines some control state and some variable of interest; the goal of syntactic slicing is to generate a syntactic slice, i.e. a syntactic subset of the program, including all statements which may affect the observation of the semantics defined by the criterion. The advantage of this approach is that the result of slicing is very easy to interpret (since, it is just a piece of code).

However, syntactic slicing presents several drawbacks. First, it may generate large slices, if the criterion depends directly or indirectly on most of the statements in the program. Second, it does not provide much information about the runtime behavior of the program being analyzed. Indeed, we expect semantic slices to characterize some set of executions and not only their trajectories in the code. Moreover, the sets of traces we intend to characterize should be defined by some semantic property (e.g., sets of initial and final states), and we expect the semantic slice to account for this property.

Fixpoint definition: Abstract invariants are usually computed by abstract interpretation of the program, i.e. computation of an abstract post-fixpoint. Therefore, we seek for a fixpoint-based definition of the semantic slice.

By definition, $\mathfrak{Slice}_{\mathbb{C}_{\mathrm{i-f}}}\langle [\![P]\!], c\rangle = \{\langle s_0, \ldots, s_n\rangle \in [\![P]\!] \mid s_0 \in \mathcal{I} \wedge s_n \in \mathcal{F}\}$. Therefore, $\mathfrak{Slice}_{\mathbb{C}_{\mathrm{i-f}}}\langle [\![P]\!], c\rangle = \overrightarrow{\mathcal{T}} \cap \overleftarrow{\mathcal{T}}$, where:

$$\overrightarrow{\mathcal{T}} = \{\langle s_0, \ldots, s_n\rangle \in [\![P]\!] \mid s_0 \in \mathcal{I}\} \qquad \overleftarrow{\mathcal{T}} = \{\langle s_0, \ldots, s_n\rangle \in [\![P]\!] \mid s_n \in \mathcal{F}\}$$

```
ℓ0 :  b := true;
ℓ1 :  if(b) {
          ...
      } else {             assumption:
          ...              b is not modified
      }                    in any branch of the if-statement
ℓ2 :  b′ := ¬b ∨ b″
ℓ3 :  input(b ∈ 𝔹);
ℓ4 :  ...
```

Figure 7.3: Backward analysis of a simple program

We proved the trace semantics semantics to be definable as a least fixpoint of some forward semantic functions $F_{\overrightarrow{P}}$ in Lemma 2.3.1. We note that $\overrightarrow{\mathcal{T}}$ is defined in a similar way as the trace semantics of P, except that we replace the set of initial states with $\mathcal{I}$. Therefore, Lemma 2.3.1 implies that $\overrightarrow{\mathcal{T}}$ is the least fixpoint of $F_{\overrightarrow{P}}$ from the set $\mathcal{T}_{\mathcal{I}}$ of traces made of a single state in $\mathcal{I}$.

Similarly, we remarked that the set of traces which terminate in some set of states can be expressed as a least fixpoint, when we introduced backward analysis in the end of Section 3.1.2: therefore, $\overleftarrow{\mathcal{T}}$ is the least fixpoint of the backward semantic function $F_{\overleftarrow{P}}$, from the set of traces $\mathcal{T}_{\mathcal{F}}$ made of a single state in $\mathcal{F}$.

As a conclusion:

$$\mathfrak{Slice}_{\mathbb{C}_{\mathrm{i-f}}}\langle [\![P]\!], c\rangle = \mathbf{lfp}_{\mathcal{T}_{\mathcal{I}}} F_{\overrightarrow{P}} \cap \mathbf{lfp}_{\mathcal{T}_{\mathcal{F}}} F_{\overleftarrow{P}}$$

7.3.2 Forward Interpreter

First, let us note that the forward fixpoint $\overrightarrow{\mathcal{T}} = \mathbf{lfp}_{\mathcal{T}_{\mathcal{I}}} F_{\overrightarrow{P}}$ can be approximated by a standard, forward static analysis as shown in Section 3.1 and Section 3.2.5.

In practice, the implementation of this analyzer follows the structure proposed in Section 3.2.5. We need the analyzer to generate an invariant for each control state, so we use in practice the analyzer with two modes *Check* and *Iter*. When considering large programs, not all local invariants can be saved in the memory, so we used refined implementation techniques, detailed in Section 7.4.4.

In the following, we write $\mathfrak{I}_0$ for the result of this forward analysis. The soundness boils down to $\overrightarrow{\mathcal{T}} \subseteq \gamma(\mathfrak{I}_0)$.

7.3.3 Backward Semantics and Backward Interpreter

Approximation of the backward fixpoint: Second, we need to perform a backward analysis, so as to approximate the second least fixpoint.

However, the backward analysis of some operations may be rather imprecise. For instance, let us consider the statement program displayed in Figure 7.3.

The backward analysis of the last statement forgets the value of b: indeed, when starting the backward analysis from point ℓ_4, there is no way to guess the value of b before its value is modified. Hence, at point ℓ_3, the backward analysis considers that b may have any boolean value. As a consequence, the backward analysis of the assignment at ℓ_2 cannot provide any information about the value of b'' at point ℓ_2, even if we know that b' is equal to **true** in the end of the program: in this case, we would expect the backward analyzer to infer that b'' is equal to **true** at point ℓ_2. Furthermore, the backward analysis executes both branches of the **if**-statement, even though only the **true** branch is executed.

In fact, this kind of issue occurs whenever a variable is assigned. Overall, a purely backward analysis would fail to compute a precise approximation of semantic slices due to these shortcomings.

As a consequence, we propose to take the results of the forward analysis into account, when performing the backward analysis. Indeed, in the above program, the forward analysis would produce a rather precise invariant, including the following predicates:

- b is equal to **true** until ℓ_4;
- the **true** branch of the conditional only is taken; any point in the **false** branch can be considered unreachable in the semantic slice.

These properties should be taken into account during the backward analysis, so as to produce precise slices.

The interpreter: The conclusion of the previous paragraph is that the backward analysis should not approximate $\overleftarrow{\mathcal{T}}$; it should rather input the approximation $\mathfrak{I}_0$ of $\overrightarrow{\mathcal{T}}$, which was computed by the forward analyzer and refine it into a new invariant, approximating the intersection $\overrightarrow{\mathcal{T}} \cap \overleftarrow{\mathcal{T}}$.

Therefore, we need a new backward interpreter which associates to any statement s a function $\overleftarrow{[\![s]\!]^\sharp} : (D^\sharp \times D^\sharp_{\mathbb{M}}) \rightarrow (D^\sharp \times D^\sharp_{\mathbb{M}})$, defined by induction over the syntax of the statements. This interpreter should input a pair made of a "global invariant" $\mathfrak{I}$ and a "local invariant" d representing a set of input states for s we wish to over-approximate the ancestors of; then it should output a pair made of a refined "global invariant" $\mathfrak{I}'$ and of an approximation d' of the input stores. Basically, $\mathfrak{I}'$ should be a refinement of $\mathfrak{I}$ (i.e., $\mathfrak{I}' \sqsubseteq \mathfrak{I}$) such that:

- for all control state ℓ in s, $\mathfrak{I}'(s)$ is derived from $\mathfrak{I}(s)$ and from the approximation of the output d;
- for all control state ℓ *not* in s, then $\mathfrak{I}'(s) = \mathfrak{I}(s)$ (the analysis does not modify the invariant outside of the analyzed statement).

In fact, such the definition of such an analyzer would be more technical and would involve more arguments:

- First, this analyzer performs side effects, whenever it *refines* the "global invariant"; therefore, it should carry out an argument specifying a mode for the analysis ($\mathcal{Check}$ or $\mathcal{Iter}$), as in Section 3.2.5.
- Second, the backward analysis should start from the set of final states $\mathcal{F}$ specified

assignment $\ell_0 : x := e; \ell_1$

$$\overleftarrow{\mathit{transfer}}_{\ell_0,\ell_1} : (d_\vdash, d_\dashv) \mapsto \overleftarrow{\mathit{assign}}(x, e, d_\vdash, d_\dashv)$$

conditional $\ell_0 : \mathbf{if}(e)\,\{\ell_0^t : s_t; \ell_1^t\}\,\mathbf{else}\,\{\ell_0^f : s_f; \ell_1^f\}\,\ell_1$

$$\begin{array}{lll}
\overleftarrow{\mathit{transfer}}_{\ell_0,\ell_0^t} : (d_\vdash, d_\dashv) \mapsto d_\vdash \sqcap d_\dashv & \text{or} & \mathit{guard}(e, \mathbf{true}, d_\vdash \sqcap d_\dashv) \\
\overleftarrow{\mathit{transfer}}_{\ell_0,\ell_0^f} : (d_\vdash, d_\dashv) \mapsto d_\vdash \sqcap d_\dashv & \text{or} & \mathit{guard}(e, \mathbf{false}, d_\vdash \sqcap d_\dashv) \\
\overleftarrow{\mathit{transfer}}_{\ell_1^t,\ell_1} = (d_\vdash, d_\dashv) \mapsto d_\vdash \sqcap d_\dashv & &
\end{array}$$

loop $\ell_0 : \mathbf{while}(e)\,\{\ell_0^b : s_t; \ell_1^b\}\,\ell_1$

$$\begin{array}{lll}
\overleftarrow{\mathit{transfer}}_{\ell_0,\ell_0^b} : (d_\vdash, d_\dashv) \mapsto d_\vdash \sqcap d_\dashv & \text{or} & \mathit{guard}(e, \mathbf{true}, d_\vdash \sqcap d_\dashv) \\
\overleftarrow{\mathit{transfer}}_{\ell_0,\ell_1} : (d_\vdash, d_\dashv) \mapsto d_\vdash \sqcap d_\dashv & \text{or} & \mathit{guard}(e, \mathbf{false}, d_\vdash \sqcap d_\dashv) \\
\overleftarrow{\mathit{transfer}}_{\ell_1^b,\ell_0} : (d_\vdash, d_\dashv) \mapsto d_\vdash \sqcap d_\dashv & &
\end{array}$$

input $\ell_0 : \mathbf{input}(x \in V); \ell_1$

$$\overleftarrow{\mathit{transfer}}_{\ell_0,\ell_1} : (d_\vdash, d_\dashv) \mapsto d_\vdash \sqcap \mathit{forget}(x, d_\dashv)$$

assertion $\ell_0 : \mathbf{assert}(e); \ell_1$

$$\overleftarrow{\mathit{transfer}}_{\ell_0,\ell_1} : (d_\vdash, d_\dashv) \mapsto d_\vdash \sqcap d_\dashv$$

Figure 7.4: Backward transfer functions

in the slicing criterion. When the backward analysis starts, s is the whole program; but $\mathcal{F}$ may specify some states *inside* the program and not necessarily in the end of the program. In this case, the backward analysis should start from the control state specified in $\mathcal{F}$, and not from the end of the program as the backward interpreter in Section 3.3.2 does.

These two issues make the definition of the backward analyzer in the style of the interpreter of Figure 3.3 very technical and not intuitive. Therefore, we provide the definition of the backward transfer functions instead (without accounting for the refinement of the "global invariant", which is done in Check mode only), in Figure 7.4. We write $\overleftarrow{\mathit{transfer}}_{\ell_0,\ell_1}$ for the backward abstract transfer function between ℓ_0 and ℓ_1. Each transfer function inputs two invariants: $d_\vdash$ stands for the invariant at the point right *before* the statement (which should be refined in the backward analysis), and $d_\dashv$ represents the invariant *after* the statement. As a consequence, the backward assignment operator is also supposed to input two arguments now.

Note that several transfer functions can be chosen, e.g., for conditions. Indeed, computing the meet of $d_\vdash$ and $d_\dashv$ seems a standard way of computing the "backward effect" of these edges; however, one may also want to enforce the condition with the help of the guard operator, so as to refine further the invariants. This issue will be considered more carefully in the next section.

The backward interpreter inputs $\mathfrak{I}_0$ and produces a refined invariant $\mathfrak{I}_1$, which satisfies the soundness condition:

Theorem 7.3.1. Soundness: backward approximation of the semantic slice.
Let us assume that $\mathfrak{I}_0$ is a sound approximation of $\overrightarrow{\mathcal{T}} \cap \overleftarrow{\mathcal{T}}$ (e.g., the invariant resulting from the forward abstract interpretation (Section 7.3.2). The invariant $\mathfrak{I}_1$ is sound:

$$\overrightarrow{\mathcal{T}} \cap \overleftarrow{\mathcal{T}} \subseteq \gamma(\mathfrak{I}_1)$$

Moreover, it refines $\mathfrak{I}_0$: $\mathfrak{I}_1 \sqsubseteq \mathfrak{I}_0$.

The backward assignment: All the transfer functions used in Figure 7.4 but the backward assignment are common, so we propose to discuss the latter in depth here.

Let us consider an assignment $l_{\text{pre}} : x := e; l_{\text{post}}$, and a pair of local invariants $d_\vdash$ and $d_\dashv$ which respectively denote the invariants available at point l_{pre} and l_{post} (after the forward analysis, $d_\vdash = \mathfrak{I}_0(l_{\text{pre}})$ and $d_\dashv = \mathfrak{I}_0(l_{\text{post}})$). Basically, we expect the analyzer to refine the local invariant $d_\vdash$, by taking into account the fact that the post-condition $d_\dashv$ should hold.

In the proofs below, we let $\rho \in \gamma_{\mathbb{M}}(d_\vdash)$; we write $v = [\![e]\!](\rho)$ and we also assume $\rho[x \leftarrow v] \in \gamma_{\mathbb{M}}(d_\dashv)$.

We distinguish boolean and scalar types for the assigned variable:

- case where x is a **boolean** variable:

$$\overleftarrow{\mathit{assign}}(x, e, d_\vdash, d_\dashv) = \left\{ \begin{array}{ll} & \mathit{guard}(e, \mathit{forget}(x, \mathit{guard}(x, d_\dashv)) \sqcap d_\vdash) \\ \sqcup & \mathit{guard}(\neg e, \mathit{forget}(x, \mathit{guard}(\neg x, d_\dashv)) \sqcap d_\vdash) \end{array} \right.$$

  Indeed, let us assume $v = \textbf{true}$. Then $\rho \in \gamma_{\mathbb{M}}(\mathit{forget}(x, \mathit{guard}(x, d_\dashv)))$, due to the hypothesis on $\rho[x \leftarrow \textbf{true}]$. Moreover, $[\![e]\!](\rho) = \textbf{true}$, so $\rho \in \gamma_{\mathbb{M}}(\mathit{guard}(e, \mathit{forget}(x, \mathit{guard}(x, d_\dashv))))$, which shows the soundness of the transfer function defined above.
- case where x is a **scalar** (i.e., integer or floating point) variable:
  1. **Linearization:** First, the expression e can be linearized into an interval linear form $\textbf{lin}(e, d_\vdash) = a_f + \sum_k a_k \cdot x_k$, where x_k is a variable and I_k is an interval (the arithmetic operators for scalars are extended to intervals). Note that this linear interval form can be computed only from $d_\vdash$ (using $d_\dashv$ would not be sound, if x appears in the right side of the expression, i.e., if x is modified by the assignment).
  2. **Refinement of interval invariants:** For any variable $y \in \{x\} \cup \{x_k \mid k\}$, we write $\mathcal{I}_y^{\text{pre}}$ (resp. $\mathcal{I}_y^{\text{post}}$) for the interval constraint for y in $d_\vdash$ (resp. $d_\dashv$).
     Our purpose is to compute a refined interval $\mathcal{I}_{x_k}^{\text{ref}}$ for any variable x_k in the right hand side of the assignment in interval linear form (if x does not appear in the right-hand side, then $\mathcal{I}_x^{\text{ref}} = \mathcal{I}_x^{\text{pre}}$).
     Let us focus on variable x_j. The soundness of linearization implies that:

$$v \in (\sum_{k \neq j} a_k \cdot \mathcal{I}_{x_k}^{\text{pre}}) + a_j \cdot \rho(x_j)$$

Hence, if $0 \notin a_j$:

$$\begin{array}{rcll} \rho(x_j) & \in & \left(v - (\sum_{k \neq j} a_k \cdot \mathcal{I}^{\mathrm{pre}}_{x_k})\right) / a_j & \text{since we can divide by } a_j \\ & \in & (\mathcal{I}^{\mathrm{post}}_x - \left(\sum_{k \neq j} a_k \cdot \mathcal{I}^{\mathrm{pre}}_{x_k})\right) / a_j & \text{since } v \in \mathcal{I}^{\mathrm{post}}_x \end{array}$$

Therefore, if we let

$$\mathcal{I}^{\mathrm{ref}}_{x_j} = \left(\left(\mathcal{I}^{\mathrm{post}}_x - (\sum_{k \neq j} a_k \cdot \mathcal{I}^{\mathrm{pre}}_{x_k})\right) / a_j\right) \cap \mathcal{I}^{\mathrm{pre}}_{x_j}$$

then, we get a sound, refined invariant for x_j before the assignment. This formula is the core of a backward assignment operator for the interval domain. If $0 \in a_j$, we cannot use this method to refine the constraint $\mathcal{I}^{\mathrm{pre}}_{x_j}$.
Note that the $d_{\vdash}$ is used not only for computing the refined intervals but also to derive the interval linear form.

3. **Other abstract domains:** other abstract domains may or may not provide any support for backward analysis; for instance, the filter domain of [Fer04b] does not. We consider the case of the octagon abstract domain; this domain provides backward transfer functions for assignment using interval linear forms [Min04b]. We should distinguish two cases:
   – If $x \in \{x_k \mid k\}$ (i.e., x appears in the right side of the interval linear form assignment):
   The default backward assignment operator provided by octagons is the function `interv_substitute_var`; it takes a linear interval form as an argument, yet it currently works in the exact case only (and behaves as a *forget* operator otherwise), so it infers new relations for variable x_k if and only if $a_k = [-1, -1]$ or $a_k = [1, 1]$. In this case, we compute $d^{\mathrm{ref}}_{\vdash}$ defined by:

   $$d^{\mathrm{ref}}_{\vdash} \quad = \quad \texttt{interv\_substitute\_var}(x, (\textstyle\sum_k (a_k \cdot x_k)) + a_f, d_{\dashv})$$

   – If $x \notin \{x_k \mid k\}$ (i.e., x does not appear in the right side of the assignment):
   The operator `interv_add_constraint` behaves like a *guard* operator, involving an interval linear form; hence, it allows for more precise handling of the expression but this will only work in the case the corresponding variables are not modified by the assignment (this is the reason why we assume here that $x \notin \{x_k \mid k\}$; the assumption that x is a sure l-value is also important here).
   In this case, we compute $d^{\mathrm{ref}}_{\vdash}$ defined by:

   $$\begin{array}{rcl} d^{\mathrm{ref}}_{\vdash} & = & \mathit{forget}(x, d_0) \\ d_0 & = & \texttt{interv\_add\_constraint}(d_1, (\sum_k (a_k \cdot x_k)) + a_f - x \leq 0) \\ d_1 & = & \texttt{interv\_add\_constraint}(d_{\dashv}, (\sum_k (a_k \cdot x_k)) + a_f - x \geq 0) \end{array}$$

The following examples show how this backward transfer functions can be applied to a few simple assignments.

Definition 7.3.1. Backward assignment; case of a boolean variable.
We extract the boolean assignment $\ell_2 : b' := \neg b \vee b''; \ell_3$ from the program displayed in Figure 7.3, and we consider the invariants (for the sake of the example, we assume that the abstract elements collect collection of non relational boolean constraints; hence, an invariant maps each boolean variable to the set of possible values for this variable):

$$\begin{array}{rcl} d_{\vdash} & = & \{b = \mathbf{true}, \ldots\} \\ d_{\dashv} & = & \{b = \mathbf{true}, b' = \mathbf{true}, \ldots\} \end{array}$$

Let us apply the formula for the boolean backward assignment:

$$\begin{array}{rcl} \mathit{guard}(\neg(\neg b \vee b''), \mathit{forget}(b', \mathit{guard}(\neg b', d_{\dashv})) \sqcap d_{\vdash}) & = & \bot \\ \mathit{guard}((\neg b \vee b''), \mathit{forget}(b', \mathit{guard}(b', d_{\dashv})) \sqcap d_{\vdash}) & = & \mathit{guard}((\neg b \vee b''), \mathit{forget}(b', d_{\dashv}) \sqcap d_{\vdash}) \\ & = & \mathit{guard}((\neg b \vee b''), \{b = \mathbf{true}, \ldots\} \sqcap d_{\vdash}) \\ & = & \mathit{guard}((\neg b \vee b''), \{b = \mathbf{true}, \ldots\}) \\ & = & \{b = \mathbf{true}, b'' = \mathbf{true}, \ldots\} \end{array}$$

As a consequence, $\overleftarrow{\mathit{assign}}(b, (\neg b \vee b''), d_{\vdash}, d_{\dashv}) = \{b = \mathbf{true}, b'' = \mathbf{true}, \ldots\}$, so that this backward transfer function is able to infer that b'' is equal to true before the assignment. We recall that we shown that this would not be possible to achieve with a transfer function which would not take $d_{\vdash}$ into account.

Definition 7.3.2. Backward assignment; domain of intervals.
We consider the assignment $x := y \cdot x + z$, with the invariants:

$$\begin{array}{rcl} d_{\vdash} & = & \{x \geq 0,\ y \in [1,2],\ z \in [1,2], \ldots\} \\ d_{\dashv} & = & \{x \in [3,4], \ldots\} \end{array}$$

Let us assume that the linearization stage converts the right-hand side $y \cdot x + z$ into $x := [1,2] \cdot x + z$ (another choice would be to turn x into an interval; however, note that the range for x in $d_{\vdash}$ is infinite so it would be a very bad choice).
Then, the backward assignment refines the range for x into $[0.5, 3]$.

Obviously, additional issue arise, when the left-hand side of the assignment is not a "sure-l-value"; for instance, if it is an array cell, which cannot be determined precisely using $d_{\vdash}$, then, not all formulae above apply (in particular, the transfer functions for octagons), so a rough approximation may need to be computed instead.

7.3.4 Combination of Forward and Backward Analyses

Need for a sequence of forward-backward analyses: In Section 7.3.1, we wrote the semantic slice as the intersection of two fixpoints; this formula served as a basis for the derivation of an abstract interpretation based approximation of the semantic slice.

However, the invariant $\mathfrak{I}_1$ (Theorem 7.3.1) may not be the optimal approximation for the semantic slice one can compute in the abstract domain. For instance, let us assume that the backward analysis reveals that no trace is going through the true branch of a conditional in the program below:

```
l  :  if(e) {
         s_t;
      } else {
         s_f;
      }
l' :  s';
l''   ...
```

Then, a refining forward analysis from $\mathfrak{I}_1$ may refine the local invariants inside s', since the possible imprecision due to the least upper bound at l' no longer occurs. Note that a further backward analysis would likely improve the results inside s_f also.

Therefore, we propose to implement a refining forward analysis and to iterate the refining forward-backward process as proposed, e.g., in [Cou78, CC92a].

Refining forward iteration: We derive the refining forward interpreter from the standard forward interpreter mentioned in Section 7.3.2 (in particular, the transfer functions and the iteration strategy are the same). The main difference is that the refining interpreter should input a global invariant $\mathfrak{I} \in D^\sharp$ approximating the semantic slice ($\mathfrak{Slice}_{\mathbb{C}_{i-f}}\langle[\![P]\!], c\rangle \subseteq \gamma(\mathfrak{I})$) to refine and use it so as to restrict each forward step.

Therefore, when analyzing a statement $l_0 : s; l_1 : \ldots$, the refining analyzer should:

- input an invariant $d_0 \in D^\sharp_{\mathbb{M}}$ for point l_0 and an invariant $\mathfrak{I} \in D^\sharp$;
- compute a refined invariant $d_1 \in D^\sharp_{\mathbb{M}}$ for point l_1;
- return the local invariant $d_1 \sqcap \mathfrak{I}(l_1)$;
- if in *Check* mode, store $d_1 \sqcap \mathfrak{I}(l_1)$ as the *refined* invariant for point l_1.

As a consequence, this refining forward analyzer is similar to the backward analyzer of Section 7.3.3, regarding to the side effects of the analysis. In the end of the analysis, it produces a refined invariant $\mathfrak{I}' \sqsubseteq \mathfrak{I}$, which is still a sound approximation of the semantic slice:

$$\mathfrak{Slice}_{\mathbb{C}_{i-f}}\langle[\![P]\!], c\rangle \subseteq \gamma(\mathfrak{I}')$$

Sequence of analyses: We now state the definition for the sequence of forward and backward analyses:

Definition 7.3.2. Refining sequence.

We define the refining sequence of invariants $(\mathfrak{I}_n)_{n\in\mathbb{N}}$.

- $\mathfrak{I}_0$ *was defined in Section 7.3.2, as the result of the initial forward analysis;*
- $\mathfrak{I}_1$ *was defined from* $\mathfrak{I}_0$ *in Section 7.3.3, as the result of the refining backward analysis; for all* $n \in \mathbb{N}$, *we let* $\mathfrak{I}_{2n+1}$ *be derived from* $\mathfrak{I}_{2n}$ *in the same way;*
- *for all* $n \in \mathbb{N}$, *we compute* $\mathfrak{I}_{2n+2}$ *by applying the refining forward analysis to* $\mathfrak{I}_{2n+1}$.

Obviously, this sequence of invariants is sound and decreasing:

Theorem 7.3.2. Properties of the refining sequence.

The sequence $(\mathfrak{I}_n)_{n\in\mathbb{N}}$ *is:*

- sound*:* $\forall n \in \mathbb{N},\ \mathfrak{Slice}_{\mathbb{C}_{i-f}}\langle [\![P]\!], c\rangle \subseteq \gamma(\mathfrak{I}_n)$;
- decreasing*:* $\forall n \in \mathbb{N},\ \mathfrak{I}_{n+1} \sqsubseteq \mathfrak{I}_n$.

Proof.

Both results follow from the properties of the refining forward and backward interpreters. □

Local iterations: The above refinement process is not optimal from the efficiency point of view. In the case of the **if** statement considered above, it amounts to completing the backward analysis of the *whole* program before doing a new forward analysis so as to refine the invariant at label l.

We might want to do *local iterations* [Gra92], that is carrying out forward and backward *local* analysis steps in a single iteration phase. For instance, Figure 7.4 displays transfer functions without and with one local iteration for conditions. Let us consider the backward analysis of the condition in the statement $l_0 : \mathbf{if}(e)\ \{l_1 \ldots$ (we consider the transfer function between l_0 and l_1):

- the standard backward transfer function is $\overleftarrow{\mathit{transfer}}_{l_0,l_0} : (d_\vdash, d_\dashv) \mapsto d_\vdash \sqcap d_\dashv$;
- if we apply the forward transfer function from l_0 to l_1, then we get a function $(d_\vdash, d_\dashv) \mapsto \mathit{guard}(e, \mathbf{true}, d_\vdash \sqcap d_\dashv)$; applying the backward function again would lead to $(d_\vdash, d_\dashv) \mapsto \mathit{guard}(e, \mathbf{true}, d_\vdash \sqcap d_\dashv) \sqcap d_\vdash = \mathit{guard}(e, \mathbf{true}, d_\vdash \sqcap d_\dashv)$ (since we assume that guard is supposed to be reductive –see Section 3.1.1).

The same local forward-backward strategy may be applied to large pieces of code; however, the choice for such strategies is very broad and most of them turn out costly.

In practice, we found that the refinement process done with an expressive, relational abstract domain (like the domain present in ASTRÉE) does not require much local iterations (except in the case of condition as described above). Carrying out iterative refinements on large blocks of code (e.g. functions) was a more efficient strategy.

7.4 Approximation of Semantic Slices

7.4.1 Extension of the Analysis

Before we can exemplify the computation of approximations for semantic slices, we need to extend the algorithm described in Section 7.3 to other semantic slicing criteria.

We use the same notations as in Section 7.3; in particular, we still consider a program P, characterized as usual by $(\mathbb{L}, \mathbb{X}, \mathbb{S}^{\mathrm{i}}, \rightarrow)$.

Execution patterns: Let us assume that some **cnt**-statements have been inserted in P and that a criterion $(\mathcal{A}, q^{\mathrm{f}}_{\mathcal{A}}) \in \mathbb{D}_{\mathbb{A}}$ is given, as in Section 7.2.3.

Intuitively, the semantic slice collects traces of the extended system $P_{\lessdot \mathcal{A} \gtrdot}$ starting from a state indexed with $q^{\mathrm{i}}_{\mathcal{A}}$, and ending in states indexed with $q^{\mathrm{f}}_{\mathcal{A}}$. Consequently, the semantic slice is defined (up to the removal of partitioning tokens) in $P_{\lessdot \mathcal{A} \gtrdot}$ by the following sets of initial and final states:

$$\begin{array}{rcl} \mathcal{I} & = & \{((l, q^{\mathrm{i}}_{\mathcal{A}}), \rho) \mid (l, \rho) \in \mathbb{S}^{\mathrm{i}}\} \\ \mathcal{F} & = & \{((l, q^{\mathrm{f}}_{\mathcal{A}}), \rho) \mid (l, \rho) \in \mathbb{S}\} \end{array}$$

Therefore, the algorithm of Section 7.3 applies to the extraction of such a slice; the main difference is that the algorithm should be applied to $P_{\lessdot \mathcal{A} \gtrdot}$.

Remark 7.4.1. More powerful partitioning domains and analyzes.
First, we note that the numeric abstractions proposed in Section 6.4 can be used instead of the automaton-based abstraction, both for the definition of the criterion and for the analysis.
Second, we mentioned, e.g., in Section 6.3.2, that the extension of the partitioning analysis into a dynamic partitioning analysis could be envisaged as well. For instance, we may assume that

- *the criterion specifies an abstraction defined by the automaton $\mathcal{A}$;*
- *the analysis starts with the abstraction defined by $\mathcal{A}$, but may refine it as suggested in Section 4.3.3 so as to compute a more precise approximation of the semantic slice.*

These extensions have not been implemented yet; however, we consider these solutions significant ideas for future work.

Input values: Let us consider a criterion $\nu \in \mathbb{C}_{\mathrm{in}}$ (constraints on the values), defined as in Section 7.2.4. For the sake of simplicity, we consider a **input**-statement l_0 : **input**$(x \in V); l_1 : \ldots$, and assume that ν only bounds the value of x at point l_1.

Let $\sigma = \langle \ldots, (l_0, \rho_0), (l_1, \rho_1), \ldots \rangle$ be a trace in the semantic slice associated to ν ($\sigma \in \mathfrak{Slice}_{\mathbb{C}_{\mathrm{in}}}\langle [\![P]\!], \nu \rangle$). Then, $\rho_1(x) \in \nu(l_1, x)$. As a consequence, all the traces in the semantic slice are also traces of the program P' derived from P by replacing the above **input** statement with l_0 : **input**$(x \in \nu(l_1, x)); l_1 : \ldots$.

Therefore, the semantic slice can be approximated in the same way as in Section 7.3, using a (slightly) different transfer function between points l_0 and l_1.

7.4.2 Examples

This section examines the examples proposed in Section 7.2, and focuses on the approximation of the corresponding semantic slices.

Definition 7.4.1. Resolution of a false alarm (Example 7.2.1 continued).
We recall the program under consideration in Figure 7.5. As previously, the semantic

```
l0 : float x, y; bool b;
l1 : input(x);
l2 : if(x > 0){
l3 :    y = x;
l4 : }else{
l5 :    y = -x;
l6 : }
l7 : b = (y > 10);
l8 : assert(b ⇒ (x < -10 ∨ 10 < x));
```

Figure 7.5: A false alarm solved

slice is defined by the following set of final states:

$$\mathcal{F} = \{(l_8, \rho) \mid \rho \in \mathbb{M},\ \rho(b) \wedge -10 \leq \rho(x) \leq 10\}$$

The table sums up the invariants computed in the first iterates of the refinement process (we perform a non-relational analysis).

point	$\mathfrak{I}_0$	$\mathfrak{I}_1$	$\mathfrak{I}_2$
l_1	$\top$	$\bot$	$\bot$
l_2	$\top$	$\bot$	$\bot$
l_7	$y \geq 0$	$\begin{cases} x \in [-10, 10] \\ y > 10 \end{cases}$	$\bot$
l_8	$\begin{cases} y \geq 0 \\ b \in \{\mathbf{true}, \mathbf{false}\} \end{cases}$	$\begin{cases} x \in [-10, 10] \\ b = \mathbf{true} \end{cases}$	$\bot$

The last column shows that the semantic slice is proved empty by the second refining iteration (second forward phase), even though the analysis is rather rough (non-relational invariants only).

$$\begin{array}{ll}
\ell_0: & \mathrm{int}\, x, y; \\
\ell_1: & x = 1; \\
\ell_2: & y = 1; \\
\ell_3: & \mathbf{while}(\mathbf{true})\{ \\
\ell_4: & \quad \mathbf{input}(x); \\
\ell_5: & \quad \mathbf{assert}(y > 0); \\
\ell_6: & \quad y = x; \\
\ell_8: & \quad \mathbf{cnt}() \\
\ell_7: & \}
\end{array}$$

(a) Modified code

$$\begin{tikzcd}
q_0 \arrow[r, "\partial_{\ell_8}"] & q_n \arrow[loop below, "\partial_{\ell_8}"]
\end{tikzcd}$$

(b) Automaton

$$\begin{array}{llll}
\nu: & (\mathbb{L} \times \mathbb{X}) & \longrightarrow & \mathcal{P}(\mathbb{V}) \\
& (\ell_4, x) & \mapsto & \{-1\} \\
& (\ell, z) \neq (\ell_4, x) & \mapsto & \mathbb{V}
\end{array}$$

(c) Input constraint

Figure 7.6: Scenario for a true error

Definition 7.4.2. Alarm pointing out a true error (Example 7.2.4 continued).
We recall the program under consideration on Figure 7.6, together with the automaton and the input function specifying the slicing criterion.
We observed in the previous subsection that this program was unsafe; if it inputs a negative value for x, it crashes at the next iteration. Therefore, the criterion:

- *restricts to the traces characterized with negative inputs at point ℓ_4;*
- *distinguishes the first iteration in the loop and the following iterations.*

The table below summarizes the result of the forward analysis (i.e., $\mathfrak{I}_0$):

point	q_0	q_n
ℓ_3	$\begin{cases} x = 1 \\ y = 1 \end{cases}$	$\begin{cases} x = -1 \\ y = -1 \end{cases}$
ℓ_5	$\begin{cases} x = -1 \\ y = 1 \end{cases}$	$\begin{cases} x = -1 \\ y = -1 \end{cases}$
ℓ_6	$\begin{cases} x = -1 \\ y = -1 \end{cases}$	$\begin{cases} x = -1 \\ y = -1 \end{cases}$

The above approximation of the semantic slice shows that the program reaches an erroneous state in the second iteration in the loop. Since this semantic slice is not empty (this program clearly has executions lasting more than one iteration and such that the inputs are negative), it proves the error scenario, we previously gave the intuition of. As a conclusion, this program is indeed flawed.

7.4.3 Use of syntactic slicing for reducing the size of programs

Program slicing: The algorithm for the extraction of semantic slices, which w presented in the previous sections suffers some significant practical weaknesses:

- it requires each analysis to save a local invariant for *each* control state, i.e., at each statement, which would result in a dramatic memory cost, when applied to large programs;
- it leads to the forward-backward analysis of the *whole* program, which would result in rather long execution times due to the analysis of the *full* program, even if only part of the program is relevant to the alarm to investigate.

Therefore, we propose to use regular, syntactic slicing techniques [Wei81, HRB90] so as to restrict the amount of code to apply the refining analyses to.

Let us assume that a program s is given, that contains a statement l_0 : **assert**(e). Whether or not an error occurs at this point depends on the variables which appear in the expression e. Therefore, the idea is to restrict to the syntactic slice defined by the control point l_0 and the variables which appear in e.

The correctness of slicing guarantees that the observation of the slice restricted to l_0, and to the variables in e *includes* the corresponding observation of the original program. As a consequence, applying the semantic slicing technique to the syntactic slice is a sound solution.

Reducing the size of slices: Even though slicing should reduce significantly the size of slices, we may want even smaller slices. Various methods serve that goal:

- First, we can use a more precise dependence analysis in order to determine a smaller slice. We investigate dependences analyses in Chapter 8, and observable dependences (Section 8.3) provide an adequate solution for restricting to the dependences, which can be observed on some *subset* of the program executions.
- Second, we may perform "aggressive slicing", i.e., remove some statement s_0, even though the alarm under investigation may depend on s_0. Of course, this solution would not be sound, since it would not take into account the effect of s_0 during the semantic slicing. Therefore, we approximate the effect of s_0. For instance, if s_0 is an assignment $x := e$, we can simply approximate s_0 with the statement **input**$(x \in \mathbb{V})$. We detail this approach in [Riv05b].

7.4.4 Implementation

The semantic slicing algorithm were implemented in the Astrée analyzer, and applied to simple programs and to large applications.

Alarm investigation process: A typical alarm investigation session proceeds as follows:

1. do a forward analysis, determine a superset of the possible errors;

2. choose an alarm to investigate; restrict to a syntactic slice [Wei81] including the alarm point;
3. define $\mathcal{I}, \mathcal{F}$, attempt to prove the alarm wrong with forward-backward refinement; otherwise, a more precise alarm context slice is found;
4. in case of failure, specialize even more the alarm context, by defining more restrictive slicing criteria
5. in case no attempt to get the analyzer to prove the emptiness of the semantic slice as we did in Example 7.4.1 succeeds, then attempt to prove the alarm corresponds to a true error by choosing a set of inputs and alarm context, in the same way as in Example 7.4.2.

Parameterization of the forward-backward analysis: The refining analysis can be applied either to the whole program or to some user-specified functions. Currently, it requires the storage of local invariants at *all* control points, in the functions the forward-backward analysis should be performed in.

The number of forward-backward iterations is also left as a parameter. The default value is 10, but we observe stabilization after 3 to 4 iterations in most cases.

We use the backward assignment operator defined in Section 7.3.3; the other abstract transfer functions are defined as usual.

More details about the implementation of the slicer will be given in Chapter 8.

Application to some large applications: We applied this technique to the alarms raised by ASTRÉE on a series of 3 early development versions of some critical embedded programs (bugs were not unlikely in the development versions).

The table below presents the results of the initial analysis. For each program, we give the size of the code, the number of functions, the analysis time and the number of alarms; each alarm was assigned a label, so that we can name it in the following discussion.

Size of the C code (lines)	70 000	226 000	400 000
Number of functions	650	1 900	2 900
Analysis time ($\mathfrak{I}_0$) in sec.	1 300	16 200	37 500
Number of alarms	4	1	0
Alarm names	a_1, a_2, a_3, a_4	a_5	-

Syntactic slicing showed that a_2 (resp. a_4) is a direct consequence of a_1 (resp. a_3); hence, we restricted to the investigation of a_1, a_3 and a_5.

The computation of a semantic slice for the corresponding dangerous states on the slices revealed rather informative conditions on the inputs. Specializing some inputs and carrying out a new, forward analysis *allowed to prove the alarms true*, thanks to an input specification as in Ex. 7.4.2.

The table below provides some data about the process: the number of input constraints is the number of points an input constraint (Definition 7.2.5) had to be specified for; the number of execution patterns corresponds to the number of criteria in $\mathbb{D}_{\mathbb{A}}$ (Definition 7.2.4). The size of the slices (number of lines, functions and variables) involved in the alarms show that a_1, a_3 were rather subtile; a_5 was much simpler. The number of additional constraints generated during the forward-backward refinement is rather difficult to express simply due to the trace partitioning, and to the use of sophisticated numerical domains; we can only mention that it is much higher than the number of variables or of program points. One forward-backward iteration necessitates a reasonable amount of resources for these slices (up to 1 min., 80 Mb).

Alarm	a_1	a_3	a_5
Size of the slice (lines)	1280	4096	244
Number of functions in the slice	29	115	8
Number of variables in the slice including: int, bool, float variables	215 $15, 60, 146$	883 $122, 553, 208$	30 $7, 11, 23$
Execution patterns	2	2	2
Input constraints	4	4	2

The only manual step is the choice of adequate execution patterns and of constraints on inputs, so as to get an error scenario; in all the above cases, these numbers are very low, which shows the amount of work for the user is very reasonable: only 4 inputs had to be chosen in the most complicated case (a_3). However each of these choices had to be made carefully, with respect to complex conditions on bit-fields and arithmetic values. The choices for the execution patterns to examine only required considering very few simple pattern criteria, akin to the automaton used for distinguishing the first iteration in Example 7.2.2 (the automaton is displayed in Figure 7.2(b)).

All errors found involve intricate floating point computations. For instance, a_5 is due to a mis-use of (interpolated) trigonometric functions, leading to a possibly negative result, causing a square root computation to fail.

Use for alarm resolution: We could also experiment the ability of the system to solve an alarm. Indeed, we considered a "legacy" alarm in the second development version, i.e. a false alarm, which was solved by a refinement of the analysis (improvement of the relational domain packing options evoked in Section 5.1.3), before semantic slicing was implemented. We disabled these relational domain packing strategies and could successfully prove the alarm false (as we did in Example 7.4.1).

Early experimental conclusions: The use of the system reduced the alarm investigation time to a few hours in the worst case we faced; the refining analyses are fully automatic and default parameters (fixed number of global forward-backward steps, no local iterations) did not have to be twicked too much to give good results. Fully manual inspection of such alarms would have required days of work and would have made the definition of an error scenario much more involved. Moreover, we could successfully classify all alarms, which means that *no false alarm remains.*

7.4.5 Comparison with related work

The idea of characterizing a set of program executions is not new.

For instance, some forms of *"conditioned" slicing* [KL88, CCL98] attack a similar problem. However, these methods are essentially based on a purely syntactic process, not only for the extraction but also for the shape of the result: a slice is defined in [Wei81] as a subset of the program statements, and these forms of slicing also produce syntactic slices.

Such forms of slicing have been employed for debugging tasks. Recent advances in this area led to the implementation of conditioned slicing tools like ConSIT [FDHH04], that could be applied to testing and software debugging [HHF+02]. However, our system is able to produce *semantic* slices, i.e., to provide global information about a set of executions instead of a mere syntactic subset of the program; this is a major advantage when investigating complex errors. The downside is that our technique relies on more sophisticated algorithms; however, syntactic slicing alone would not help significantly the alarm inspection process in Astrée.

The *search for counter-examples* and *automatic refinement* has long been a motivation in the model-checking-based systems, such as [CGJ+00, BNR03, PHR04, GRS00]. In particular, the automatic refinement process plays a great role in the determination of the set of predicates (i.e. abstract domain) needed for a precise analysis [BMMR01]. Our goal is to bring such methods in static analyzers like Astrée for a different purpose, i.e. to solve the few, subtile alarms, after an already very precise analysis [BCC+03a] (the construction of the domain requires no internal refinement process).

Forward-backward analysis schemes have been applied, e.g. in [Jea03], to the inference of safety properties. Some static analysis systems have been extended with counter-examples search facilities: [GJJM03] relies on random test generation; [Ere04] uses a symbolic under-approximation of erroneous traces and theorem proving. The main difference is that we chose to start with an over-approximation of erroneous traces until conditions on inputs are precise enough so that a counter-example could be found since the search space for counter-examples was huge in our case, due to the size of the programs. For instance, the systematic exploration of paths as in [Ere04] over length above 1 000, with hundreds of variables would not work. Moreover, we allow abstract error scenario to be tested unlike [GJJM03, Ere04]: this reduces the amount of input constraints to fix to a minimum. On the other hand, at this time, we still do not perform the automatic generation of counter-examples, which is left as a future work.

7.4.6 Future work

At the time of the writing, we have plans for extending the framework for semantic slicing presented in this chapter, in addition to the improvement of the current implementation, which is still not really usable by a non-specialist.

Allow for automatic refinement of criteria: A first, very important area for future work would consist in refining the criteria in a semi-automatic or automatic way. For instance, we would like to allow some kind of dynamic partitioning of the execution pattern criteria.

The two main difficulties to solve to achieve that goal are:

- the **choice of refinements:** for instance, choosing sensible refinements for the automaton given in the "execution pattern" criterion (Section 7.2.3) from the numerical invariants is a difficult task, which requires efficient strategies to be found;
- the **definition of a widening for the domain of criteria** is also a tedious issue, even if tree schemata may provide the basis for some solution (Section 6.3.2).

Automatizing the search for error scenarios: Second, the automatic generation for error scenarios is a very challenging and important goal. The semantic slicing exposed here should help to determine precise *over*-approximation for erroneous traces; however, a more convincing result would be a counter-example, which would precisely tell the user what the bug is.

For instance, [Ere04] collects and then solves symbolic constraints so as to find a counter-example. We plan to attempt to implement similar techniques in the near future.

Obviously, the generation for counter-examples would require the computation of an under-approximation of the erroneous traces; yet, such an under-approximation may turn out to contain fictitious traces only, which is a major issue.

Chapter 8

Computation of Abstract Dependences

We study various forms of dependences, so as to localize the cause for some behaviors of programs. In particular, we wish to track the cause for erroneous behaviors, such as division by 0, overflows...

We choose to set-up definitions of dependences, which are close to the common definition of non-interference [GM82], so as to start with a *semantic* notion of dependence, which is rather more adapted for defining extensions than classical syntactic definitions. We state this framework in Section 8.2.

Then we propose several extension of this classical notion of dependence. First, we define *observable dependences* in Section 8.3, by restricting to a subset of the traces of a program, i.e., to a semantic slice. Second, we introduce *Abstract dependences* in Section 8.4, as a way to relate abstract properties in programs as well. We provide algorithms for approximating each form of dependence.

Then, we discuss informally the extraction of slices in Section 8.5, using the various forms of dependences, which we introduced in the chapter.

We conclude the chapter in Section 8.6 with a short case study, a comparison with related work and an outline of the main perspectives for continuing this work.

8.1 Motivation

We introduced *semantic slicing* in Chapter 7 as a means to extract effectively a subset of the traces of a program, so as to attempt to solve the alarms generated by a static analysis. In particular, semantic slicing can prove an alarm false, by proving that no real execution causes the corresponding runtime error. It can also be helpful in producing and checking an error scenario, i.e., a trace resulting in a runtime error. Therefore, semantic slicing is helpful in the alarm investigation process.

However, this technique does not solve all the issues, which arise, when trying to understand the origin of an alarm:

- the **origin of imprecision or errors is not found**: semantic slicing only provides refined conditions for an error to happen; yet, finding what part of the program may cause an error is a completely different issue, which we definitely want to address.
- the **amount of data to inspect may still be cumbersome**: indeed, the invariants computed during semantic slicing may contain a huge amount of relevant information, and a user would expect some help about what to look at first.
- the **synthesis of semantic slicing criteria is still not automatic**: we did not provide any automatic way to guess useful slicing criteria in Chapter 7; however, this might turn out a difficult task –especially for non-experienced users.

Obviously, the first point present some similarities with a dependence problem. In fact, it is very difficult to define what the "cause" for an error is. In practice, a programmer investigating a bug attempts to reconstitute the sequence of events, which caused a failure to occur: the investigation starts from the point where the error occurs; then, the origin values of the variables affecting the error should be checked and so on recursively. The manual alarm investigation technique proceeds similarly, by looking at invariants. This approach can clearly be assimilated to a kind of dependence analysis, starting from the error or alarm point.

The second point, i.e., choosing what part of the invariants should be investigated first also reduces to the resolution of a problem of dependences: indeed, it is very natural to focus first on the dependence of the variables incriminated in the alarm, and to focus on what the error condition depends on.

The third point does not reduce straightforwardly to a dependence problem. However, we can distinguish the following issues:

- the *initial criterion* should be determined by an alarm raised by the analyzer; more precisely, it should specialize the analysis to a case where the error does occur;
- the *refined criteria* should refine the semantic slice, and try to improve the characterization of the traces leading to an error; moreover, it should refine first the analysis of the statements encountered before the alarm, and which impact the variables involved in the alarm, so as to provide a better understanding of the immediate context of the alarm first.

Clearly, the dependences from the error condition, in the semantic slice should be useful in the case of refined criteria, since the dependences computed from the alarm point should tell what part to refine first.

Syntactic slicing [Wei81] is based on dependence analysis as well; however, it focus on the extraction of *all* the statements the criterion depends on. By contrast, we would be interested in *dependence chains* rather than the whole slice, even though the slice may also be useful in a second step.

However, program slicing techniques [Wei81, HRB90] usually rely on *syntactic dependences*: indeed, the dependences collected in the slicing process are characterized by "def-use" conditions. We may wish to focus on more informative dependences, so as to track and characterize the causes for errors. We illustrate this issue in the following example.

Definition 8.1.1. Semantic slice and dependences.
Let us consider the program P in Figure 8.1(a); in particular, we focus on the semantic slice defined by the constraints displayed in Figure 8.1(b). Intuitively, the criterion specifies some initial condition (e.g., a condition on the dynamic inputs of the program) and it aims at studying the traces which result in a large value of y.

l_0	$\textbf{if}(x > 5)\{$	Initial condition (l_0):
l_1	$\quad y = 1\,000 \star x;$	$x \in [0, 10]$
l_2	$\}\,\textbf{else}\,\{$	$y \in [0, 5]$
l_3	$\quad y = y + z;$	$z \in [-4, 15]$
l_4	$\}$	Final condition (l_5):
l_5	$\ldots$	$y \geq 1\,000$

(a) Code

(b) Semantic slicing criterion

Figure 8.1: Dependence analysis for alarm investigation

We provide in the table below a synthetic characterization of the semantic slice defined by the criterion (we use interval invariants).

Point	*Invariant*		
	x	y	z
l_0	$[0, 10]$	$[0, 5]$	$[-4, 15]$
l_1	$[6, 10]$	$[0, 5]$	$[-4, 15]$
l_2	$[6, 10]$	$[6\,000, 10\,000]$	$[-4, 15]$
l_3	$\perp$	$\perp$	$\perp$
l_4	$\perp$	$\perp$	$\perp$
l_5	$[6, 10]$	$[6\,000, 10\,000]$	$[-4, 15]$

Obviously, no trace in the semantic slice goes through the $\Delta = \Delta$ branch of the conditional, since this branch would only generate small values for y under the input condition given above. Moreover, we can see in the semantic slice that the first occurrence of a large value in the program occurs at point l_2, after the assignment $y = 1\,000 \star x$.
As a consequence, we intend to define a dependence analysis such that:

- *the dependences induced in the* **false** *branch are not collected;*
- *the dependence from (l_5, y) to the assignment $y = 1\,000 \star x$ is more important, hence should be collected in priority.*

Before we tackle the definition of dependences fulfilling the requirements stated in Example 8.1.1, we need to choose a framework for expressing dependences, which is the goal of the next section.

8.2 Notion of Dependences and Approximation

First, we set up a framework for reasoning about dependences. The notions and notations used in this section will be used thoroughly further in the chapter.

The definitions of dependences we are going to set up are based on denotational abstractions; as a consequence, we assume that $[\![P]\!]$ is the "strongly closed" version of the semantics of programs defined in Section 3.2.1. In fact, we go even further and assume that $[\![P]\!]$ collects *all* traces of the transition system P starting from *any* state, i.e., we let $[\![P]\!] = \mathbf{lfp}_{\emptyset} F_P$, where:

$$\begin{array}{rlll} F_P : & \mathcal{P}(\Sigma) & \longrightarrow & \mathcal{P}(\Sigma) \\ & \mathcal{E} & \mapsto & \{\langle s \rangle \mid s \in \mathbb{S}\} \cup \{\langle s_0, \ldots, s_n, s_{n+1} \rangle \in \Sigma \mid \langle s_0, \ldots, s_n \rangle \in \mathcal{E} \wedge s_n \rightarrow s_{n+1}\} \end{array}$$

We will refine this assumption in Section 8.3; indeed the definition of observable traces will allow to restrict –among others– to the traces starting from some initial state.

8.2.1 Dependences induced by a function

Defining dependences: Dependences have a nicer formulation when considering functions instead of mere traces: an output depends on the inputs that may affect its result. Hence, we start with a study of the dependences expressed on functions. Later, we shall use the abstraction of traces into functions (Section 3.2).

Definition 8.2.1. Dependences.

Let $\phi \in \mathfrak{Den}$, $x_0, x_1 \in \mathbb{X}$. We say that ϕ induces a dependence *of x_1 on x_0 if and only if there exist $\rho_0 \in \mathbb{M}$, $v_a, v_b \in \mathbb{V}$ such that $\phi(\rho_a)(x_1) \neq \phi(\rho_b)(x_1)$ where $\rho_i = \rho_0[x_0 \leftarrow v_i]$. Such a dependence is written $x_1 \overset{\phi}{\rightsquigarrow} x_0$ (or $x_1 \rightsquigarrow x_0$ when there is no ambiguity about the function ϕ).*

Intuitively, there is a dependence of x_1 on x_0 if a single modification of the input value of x_0 may result in a different result for x_1. In other words, there is a dependence of x_1 on x_0 if and only if the observation of output value for x_1 gives any information about the input value for x_0.

Definition 8.2.1. Dependences of functions.

Let $x, y \in \mathbb{X}$. Let us consider the function $\phi \in \mathfrak{Den}$ defined by

$$\phi(\rho) = \left\{ \begin{array}{ll} \{\rho[y \leftarrow x]\} & \textit{if } \rho(b) = \mathbf{true} \\ \emptyset & \textit{if } \rho(b) = \mathbf{false} \end{array} \right.$$

Then, if $\rho_0 \in \mathbb{M}$, and $z \in \mathbb{X}$, $\phi(\rho_0[b \leftarrow \mathbf{false}])(z) = \emptyset \neq \phi(\rho_0[b \leftarrow \mathbf{true}])(z)$; hence, $z \overset{\phi}{\rightsquigarrow} b$. Similarly, we would show that $y \rightsquigarrow x$.
Last, if $z \in \mathbb{X} \setminus \{y\}$, we could prove that $z \overset{\phi}{\rightsquigarrow} z$, and that ϕ has no other dependence.

Dependences and non-secrecy: Definition 8.2.1 presents some deep similarities with the notion of non-interference (or secrecy) [GM82], which is commonly used in the area of security. In this setting, the set of variables $\mathbb{X}$ is usually partitioned into two parts:

- the "*low*" variables ($\mathbb{X}^{\mathrm{L}}$) should be public (their value may be read by anyone);
- the "*high*" variables ($\mathbb{X}^{\mathrm{H}}$) should be private: only authorized users should access them; moreover, other users should not be able to derive any information about high variables, e.g., by observing low variables.

We assume that such a partition is given; then, secrecy usually boils down to:

Definition 8.2.2. Secrecy.

Let $\phi \in \mathfrak{Den}$. We say that ϕ is secure if and only if the following condition holds:

$$\forall \rho_0, \rho_1 \in \mathbb{M},\ \big(\forall x \in \mathbb{X}^{\mathrm{L}},\ \rho_0(x) = \rho_1(x)\big) \Longrightarrow \forall x \in \mathbb{X}^{\mathrm{L}},\ \phi(\rho_0)(x) = \phi(\rho_1)(x)$$

Intuitively, if ϕ is secure, then observing the low outputs does not provide any information about the high inputs: indeed, if two inputs may only differ in the value of the high variables, then the resulting outputs have the same low observation.

Other authors used similar formalisms in order to describe, e.g., information flows in programs [Den76, DD77].

By contrast, in the case of the definition of dependences (Definition 8.2.1), we can note two important differences:

- the partition of $\mathbb{X}$ in low and high variables is not the same for the inputs and for the outputs, as summarized in the table below (we keep the notations of Definition 8.2.1):

	Input	Output
High	$\mathbb{X}^{\mathrm{H}}_{\mathrm{in}} = \{x_0\}$	$\mathbb{X}^{\mathrm{H}}_{\mathrm{out}} = \mathbb{X} \setminus \{x_1\}$
Low	$\mathbb{X}^{\mathrm{L}}_{\mathrm{in}} = \mathbb{X} \setminus \{x_0\}$	$\mathbb{X}^{\mathrm{L}}_{\mathrm{out}} = \{x_1\}$

- we say that there is a dependence if we can observe a modification of the value of x_0 before applying ϕ by observing the value of x_1 after: therefore, the existence of a dependence is the opposite of secrecy (a function is secure if there is no dependence).

Consequently, our notion of dependence is a equivalent to a form of non-secrecy. The reason why we adopt such a definition is that we wish to start with a semantic definition of what a dependence is, so as to be able to design various extensions and refinements later; the syntactic definitions traditionally used in slicing would not allow this to be done.

Dependence abstraction: We now define the set of dependences of a function:

Definition 8.2.3. Dependence set.

We use the same notations as in Definition 8.2.1. We let the dependence set $\mathfrak{D}_{\mathrm{f}}[\phi]$ *of ϕ be the set of dependences induced by ϕ:*

$$\mathfrak{D}_{\mathrm{f}}[\phi] = \{(x_0, x_1) \mid x_1 \overset{\phi}{\rightsquigarrow} x_0\} \in \mathfrak{Dep}_{\mathrm{f}}$$

We write $\mathfrak{Dep}_{\mathrm{f}} = \mathcal{P}(\mathbb{X}^2)$, so that $\mathfrak{D}_{\mathrm{f}}[\phi] \in \mathfrak{Dep}_{\mathrm{f}}$.

Definition 8.2.2. Non-determinism and dependences.

We let $x_1 \in \mathbb{X}$, and ϕ be the function defined by $\phi : \rho \mapsto \{\rho[x_1 \leftarrow v] \mid v \in \mathbb{V}\}$. Intuitively, ϕ represents the semantics of a random statement.
Let x_0 be any variable and ρ be a store. Then, $\forall v \in \mathbb{V},\ \phi(\rho[x_0 \leftarrow v])(x_1) = \mathbb{V}$. Therefore, $(x_0, x_1) \notin \mathfrak{D}_{\mathrm{f}}[\phi]$.
Let $x_0 \in \mathbb{X},\ x_0 \neq x_1$. Then, we can check straightforwardly that $x_0 \overset{\phi}{\rightsquigarrow} x_0$, since $\phi(\rho)(x_0) = \{\rho(x_0)\}$.
Hence,

$$\mathfrak{D}_{\mathrm{f}}[\phi] = \{(x_0, x_0) \mid x_0 \in \mathbb{X} \wedge x_0 \neq x_1\}$$

We derive an abstraction for sets of elements of $\mathfrak{Den}$ from the function $\phi \mapsto \mathfrak{D}_{\mathrm{f}}[\phi]$:

Definition 8.2.4. Dependence abstraction.

We consider $\mathfrak{Dep}_{\mathrm{f}} = \mathcal{P}(\mathbb{X} \times \mathbb{X})$, with the usual set inclusion ordering. Then, we have a Galois connection:

$$(\mathcal{P}(\mathfrak{Den}), \subseteq) \underset{\alpha_{\mathfrak{D}}}{\overset{\gamma_{\mathfrak{D}}}{\leftrightarrows}} (\mathfrak{Dep}_{\mathrm{f}}, \subseteq)$$

where:

$$\begin{array}{rlll} \alpha_{\mathfrak{D}} : & \mathcal{P}(\mathfrak{Den}) & \rightarrow & \mathfrak{Dep}_{\mathrm{f}} \\ & \Phi & \mapsto & \{(x_0, x_1) \mid \exists \phi \in \Phi,\ \overset{x_1}{\rightsquigarrow} \phi x_0\} \\ \gamma_{\mathfrak{D}} : & \mathfrak{Dep}_{\mathrm{f}} & \rightarrow & \mathcal{P}(\mathfrak{Den}) \\ & \mathcal{D} & \mapsto & \{\phi \in \mathfrak{Den} \mid \mathfrak{D}_{\mathrm{f}}[\phi] \subseteq \mathcal{D}\} \end{array}$$

The proof that $(\alpha_{\mathfrak{D}}, \gamma_{\mathfrak{D}})$ define a Galois-connection is straightforward.

Please note that a dependence set is an abstraction of a *set of* functions and *not* for a single function. In particular, the function $\phi \mapsto \mathfrak{D}_{\mathrm{f}}[\phi]$ is not even monotone, as stated in the following remark, so that it is not possible to define a Galois connection, where $\mathfrak{D}_{\mathrm{f}}[.]$ would be the abstraction function.

Remark 8.2.1. Non monotonicity.

The function $\phi \mapsto \mathfrak{D}_{\mathrm{f}}[\phi]$ is non monotone*: $\exists \phi, \phi',\ \forall \rho \in \mathbb{M},\ \phi(\rho) \subseteq \phi'(\rho) \wedge \mathfrak{D}_{\mathrm{f}}[\phi] \nsubseteq \mathfrak{D}_{\mathrm{f}}[\phi']$. For instance, the upper element of $\mathfrak{Den}$ is $\phi_{\top} : \rho \mapsto \mathbb{M}$; and, $\mathfrak{D}_{\mathrm{f}}[\phi_{\top}] = \emptyset$, so proving the non monotonicity of the $\mathfrak{D}_{\mathrm{f}}[.]$ operator reduces to finding a function which has at least one dependence. This is possible if the number of elements of $\mathbb{V}$ is greater than 2, which is always the case in practice.*

Approximation of composition: We noted in Section 3.2 that the $\circ$ operator is the counterpart for the concatenation of statements, execution paths... Therefore, we propose to determine the dependences of the composition of functions: if $\phi_0, \phi_1 \in \mathfrak{Den}$, then we wish to derive an approximation for $\mathfrak{D}_{\mathrm{f}}[\phi_1 \circ \phi_0]$. The $\boxdot$ operator simply composes dependences:

Definition 8.2.5. Junction of dependence sets.

Let $\mathfrak{D}, \mathfrak{D}' \in \mathfrak{Dep}_{\mathrm{f}}$. We define the junction *of $\mathfrak{D}$ and $\mathfrak{D}'$ denoted with $\mathfrak{D} \boxdot \mathfrak{D}'$ by*

$$\mathfrak{D} \boxdot \mathfrak{D}' = \{(x, x'') \in \mathbb{X}^2 \mid \exists x' \in \mathbb{X},\ (x, x') \in \mathfrak{D} \wedge (x', x'') \in \mathfrak{D}'\}$$

Lemma 8.2.1. Monotonicity of the junction operator.

The operator $\boxdot$ is monotone: *if $\mathfrak{D}_0, \mathfrak{D}'_0, \mathfrak{D}_1, \mathfrak{D}'_1 \in \mathfrak{Dep}_{\mathrm{f}}$ are such that $\mathfrak{D}_0 \subseteq \mathfrak{D}'_0$ and $\mathfrak{D}_1 \subseteq \mathfrak{D}'_1$, then $\mathfrak{D}_0 \boxdot \mathfrak{D}_1 \subseteq \mathfrak{D}'_0 \boxdot \mathfrak{D}'_1$.*

Proof.

Let $(x, x'') \in \mathfrak{D}_0 \boxdot \mathfrak{D}_1$. Then, there exists $x' \in \mathbb{X}$, such that $(x, x') \in \mathfrak{D}_0$ and $(x', x'') \in \mathfrak{D}_1$. By assumption, $\mathfrak{D}_0 \subseteq \mathfrak{D}'_0$, so $(x, x') \in \mathfrak{D}'_0$; similarly, $(x', x'') \in \mathfrak{D}'_1$. As a consequence, $(x, x') \in \mathfrak{D}'_0 \boxdot \mathfrak{D}'_1$. $\square$

The operator $\boxdot$ over-approximates the dependences of the composition of functions:

Theorem 8.2.2. Composition of dependences –approximation.

The operator $\boxdot$ is a sound approximation for ";" (or $\circ$); that is, if $\phi_0, \phi_1 \in \mathfrak{Den}$ such that,

$$\mathfrak{D}_{\mathrm{f}}[\phi_1 \circ \phi_0] \subseteq \mathfrak{D}_{\mathrm{f}}[\phi_0] \boxdot \mathfrak{D}_{\mathrm{f}}[\phi_1]$$

Proof.

We write $\mathfrak{D}$ for $\mathfrak{D}_{\mathrm{f}}[\phi_0] \boxdot \mathfrak{D}_{\mathrm{f}}[\phi_1]$; we let $\phi = \phi_1 \circ \phi_0$. Let $x_0, x_2 \in \mathbb{X}$. Let us assume that $(x_0, x_2) \notin \mathfrak{D}$ and show that $\neg(x_2 \overset{\phi}{\rightsquigarrow} x_0)$.

Since $(x_0, x_2) \notin \mathfrak{D}$, $\forall x_1 \in \mathbb{X}$, $\Big(\neg(x_1 \overset{\phi_0}{\rightsquigarrow} x_0) \vee \neg(x_2 \overset{\phi_1}{\rightsquigarrow} x_1)\Big)$. Let $\rho \in \mathbb{M}$, $v, v' \in \mathbb{V}$. We let:

$$\begin{array}{rclrcl}
\rho_0 & = & \rho[x_0 \leftarrow v] & \rho'_0 & = & \rho[x_0 \leftarrow v'] \\
P_1 & = & \phi_0(\rho_0) & P'_1 & = & \phi_0(\rho'_0) \\
P_2 & = & \phi_1(P_1) & P'_2 & = & \phi_1(P'_1)
\end{array}$$

We intend to show that $\phi(\rho_0)(x_2) = \phi(\rho'_0)(x_2)$, that is $P_2(x_2) = P'_2(x_2)$.

The execution of ϕ_0 modifies the value of at most a finite number of variables. Let V be the set of modified variables by executing ϕ_0 either from ρ_0 or from ρ'_0 and W be the set $\{x \in \mathbb{X} \mid P_1(x) \neq P'_1(x)\}$.

Clearly, W is finite. Moreover, if $x_1 \in W$, then $x_1 \overset{\phi_0}{\rightsquigarrow} x_0$; hence, $\neg(x_2 \overset{\phi_1}{\rightsquigarrow} x_1)$.
We prove straightforwardly by induction on $\mathbf{Card}(W)$ that:

$$\forall Q_1, Q'_1 \in \mathcal{P}(\mathbb{M}),\quad \left.\begin{array}{l} W = \{x \in \mathbb{X} \mid Q_1(x) \neq Q'_1(x)\} \\ x_1 \in W \Longrightarrow \neg(x_2 \overset{s_1}{\rightsquigarrow} x_1) \end{array}\right\} \Longrightarrow \phi_1(Q_1)(x_2) = \phi_1(Q'_1)(x_2)$$

- if $\mathbf{Card}(W) = 0$, then, $Q_1 = Q'_1$, so the result is obvious;
- if $\mathbf{Card}(W) = n + 1$ and the property holds for n, then we can pick up an element $x_1 \in W$ and let $W' = W \setminus \{x_1\}$. We define $Q''_1 = \{\rho_1[x_1 \leftarrow \rho'_1(x_1)] \mid \rho_1 \in Q_1, \rho'_1 \in Q'_1\}$. Then:

$$\begin{array}{rcll} \phi_1(Q_1)(x_2) & = & \phi_1(Q''_1)(x_2) & \text{since } \neg(x_2 \overset{\phi_1}{\rightsquigarrow} x_1) \text{ and } \forall x \in \mathbb{X} \setminus \{x_1\},\ Q_1(x) \neq Q''_1(x) \\ & = & \phi_1(Q'_1)(x_2) & \text{by induction hypothesis, and since } \mathbf{Card}(W') = n \end{array}$$

Therefore, the property applies to the above set W and $P_2(x_2) = P'_2(x_2)$. $\square$

8.2.2 Dependences induced by a set of traces

In the following a dependence observed on a set of traces states that "the value of variable x_1 at point l_1 depends on the value of variable x_0 at point l_0". We derive such dependences from the dependences induced by a function obtained by applying to $\mathcal{E}$ either of the abstractions, we introduced in Section 3.2.

Definition: First, we define the dependences between two fixed control states:

Definition 8.2.6. From-to Dependences.
Let $\mathcal{E}$ be a set of traces. For any pair of points $l_0, l_1 \in \mathbb{L}$, the "from-to" dependence set *$\mathfrak{D}_f[\mathcal{E} \mid l_0, l_1]$ is defined by:*

$$\mathfrak{D}_f[\mathcal{E} \mid l_0, l_1] = \mathfrak{D}_{\mathrm{f}}[\alpha_{t\mathcal{F}[l_0,l_1]}(\mathcal{E})]$$

By extension, if s is a statement, then: $\mathfrak{D}_f[s \mid l_0, l_1] = \mathfrak{D}_f[[\![s]\!] \mid l_0, l_1] = \mathfrak{D}_{\mathrm{f}}[\alpha_{t\mathcal{F}[l_0,l_1]}([\![s]\!])]$.

The dependences for the whole set of traces collect all from-to dependences:

Definition 8.2.7. Dependences.
The dependence set of $\mathcal{E}$ is:

$$\mathfrak{D}_t[\mathcal{E}] = \{((l_0, x_0), (l_1, x_1)) \mid (x_0, x_1) \in \mathfrak{D}_f[\mathcal{E} \mid l_0, l_1]\} \in \mathfrak{Dep}_{\mathrm{t}}$$

By extension, $\mathfrak{D}_t[s] = \mathfrak{D}_t[[\![s]\!]]$.

We can also restrict the observation of dependences to a path:

Definition 8.2.8. Dependences along a path.

Let $l_\vdash, l_\dashv \in \mathbb{L}$ and $p \in \mathcal{P}(l_\vdash, l_\dashv)$. We let $\mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p]$ be the dependence sets induced by $\mathcal{E}$, restricted to the path p by taking into account the traces on the path p only:

$$\mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p] = \mathfrak{D}_{\mathrm{f}}[\alpha_{p\mathcal{F}[p]}(\llbracket \mathcal{E} \rrbracket)]$$

Path decomposition: In particular, we note that the set of dependences along all paths between a pair of points partition the from-to dependences between these two points; this result will play a significant role in the definition of a computable approximation for dependences:

Theorem 8.2.3. Approximating the from-to dependences.

Let $x_0, x_1 \in \mathbb{X}$ and $l_0, l_1 \in \mathbb{L}$. If $(x_0, x_1) \in \mathfrak{D}_f[\mathcal{E} \mid l_0, l_1]$, then there exists $p \in \mathcal{P}(l_0, l_1)$, such that $(x_0, x_1) \in \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p]$:

$$\mathfrak{D}_f[\mathcal{E} \mid l_0, l_1] \subseteq \bigcup \{\mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p] \mid p \in \mathcal{P}(l_0, l_1)\}$$

Proof.

We show the contraposition: we assume that $\forall p \in \mathcal{P}(l_0, l_1)$, $(x_0, x_1) \notin \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p]$ and we show that $(x_0, x_1) \notin \mathfrak{D}_f[\mathcal{E} \mid l_0, l_1]$.

Let $\rho \in \mathbb{M}$ and $v, v' \in \mathbb{V}$. We intend to show that $\alpha_{t\mathcal{F}[l_0,l_1]}(\mathcal{E})(\rho[x_0 \leftarrow v])(x_1) = \alpha_{t\mathcal{F}[l_0,l_1]}(\mathcal{E})(\rho[x_0 \leftarrow v'])(x_1)$. Let us note that:

$$\begin{aligned} &\alpha_{t\mathcal{F}[l_0,l_1]}(\mathcal{E})(\rho[x_0 \leftarrow v])(x_1) \\ &\quad = \bigcup \{\alpha_{p\mathcal{F}[p]}(\mathcal{E})(\rho[x_0 \leftarrow v])(x_1) \mid p \in \mathcal{P}(l_0, l_1)\} \quad \text{because of lemma 3.2.1} \end{aligned}$$

The assumption $(x_0, x_1) \notin \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p]$ implies that $\alpha_{p\mathcal{F}[p]}(\mathcal{E})(\rho[x_0 \leftarrow v])(x_1) = \alpha_{p\mathcal{F}[p]}(\mathcal{E})(\rho[x_0 \leftarrow v'])(x_1)$ for any path $p \in \mathcal{P}(l_0, l_1)$. Hence,

$$\begin{aligned} \alpha_{t\mathcal{F}[l_0,l_1]}(\mathcal{E})(\rho[x_0 \leftarrow v])(x_1) &= \bigcup \{\alpha_{p\mathcal{F}[p]}(\mathcal{E})(\rho[x_0 \leftarrow v'])(x_1) \mid p \in \mathcal{P}(l_0, l_1)\} \\ &= \alpha_{t\mathcal{F}[l_0,l_1]}(\mathcal{E})(\rho[x_0 \leftarrow v'])(x_1) \qquad \text{(as above)} \end{aligned}$$

This concludes the proof. □

We note that the approximation of $\mathfrak{D}_f[\mathcal{E} \mid l_0, l_1]$ given in Theorem 8.2.3 is usually strict, and might affect the precision of analyses, as shown in the following example:

Definition 8.2.3. Dependences in a program.
Let us consider the program P below:

$$\begin{array}{ll} l_0 : & \mathbf{if}(b)\,\{ \\ l_1 : & \qquad x = 4; \\ l_2 : & \}\,\mathbf{else}\,\{ \\ l_3 : & \qquad x = 4; \\ l_4 : & \} \\ l_5 : & \ldots \end{array}$$

Then, there are two paths $p_\mathrm{t}, p_\mathrm{f}$ (one path through each branch of the conditional) from l_0 to l_5, so Theorem 8.2.3 gives the approximation: $\mathfrak{D}_f[P \mid l_0, l_5] \subseteq \mathfrak{D}_{\mathcal{F}}[P \mid p_\mathrm{t}] \cup \mathfrak{D}_{\mathcal{F}}[P \mid p_\mathrm{f}]$. However,

- *the same value is assigned to x whatever the path, so $\neg((l_5, x) \overset{P}{\rightsquigarrow} (l_0, b))$;*
- *$\alpha_{p\mathcal{F}[p_\mathrm{t}]}([\![P]\!]) = [\![\lfloor b \ ?\ \lfloor x \leftarrow 4 \rfloor \ \mid\ \Box\ \rfloor]\!]$, hence $(b, x) \in \mathfrak{D}_{\mathcal{F}}[P \mid p_\mathrm{t}]$ (and the same for $\mathfrak{D}_{\mathcal{F}}[P \mid p_\mathrm{f}]$).*

As a consequence, $\mathfrak{D}_{\mathcal{F}}[P \mid p_\mathrm{t}] \cup \mathfrak{D}_{\mathcal{F}}[P \mid p_\mathrm{f}]$ is a strict *over-approximation of $\mathfrak{D}_f[P \mid l_0, l_5]$. In fact, this example reveals even worse imprecisions: for instance, if $y \in \mathbb{X} \setminus \{b, x\}$, then $(b, y) \in \mathfrak{D}_{\mathcal{F}}[P \mid p_\mathrm{t}]$. Such imprecisions will be addressed in the Section 8.2.4.*

Errors and non-termination: Let us consider the program l_0 : $\mathbf{assert}(b); l_1$. If b is false, then the program crashes (the execution stops), so that the image of the denotational semantics of this program is $\emptyset$; if b is true, then it behaves like the identity function. As a consequence, for all $x \in \mathbb{X}$, $(l_1, x) \rightsquigarrow (l_0, b)$. We may not want to include such dependences. Either this would amount to include ways too many dependences, or these dependences would not have a practical interpretation, if we wish to understand the way $x \neq b$ is computed.

Note that the same issue occurs with non-termination: if we consider l_0 : $\mathbf{while}(b)\{\}; l_1$, if b is true, the execution never reaches point l_1.

The common solution to this issue consists in using a "lazy semantics" [CF89], allowing erroneous executions to continue, with an "error-flag" turned on; similarly looping execution can continue after diverging, with only the ultimately constant variables well-defined after the point of divergence. The presentation used in [CF89] is denotational, but other authors [GM03] also proposed lazy versions of trace semantics (roughly, they allow "transfinite traces").

Basically, our framework works in both cases (i.e., for standard semantics as well as for lazy semantics).

8.2.3 Approximation of dependences

We address in this subsection the computation of an approximation of the dependence set introduced in Definition 8.2.7.

Local dependences: In a real program, the dependences induced by each statement can be determined pretty easily by local rules.

In our present set-up, this local description of the dependences of the program can be defined by an approximation of the dependences induced by one-step transitions.

Definition 8.2.9. Local dependences.

We define the local dependences *induced by a set of traces $\mathcal{E}$ as the dependences that can be observed on paths of length* 1*:*

$$\mathfrak{D}_{\text{loc}} = \{((\ell_0, x_0), (\ell_1, x_1)) \in (\mathbb{L} \times \mathbb{X})^2 \mid \ell_0, \ell_1 \in \mathbb{L} \wedge (x_0, x_1) \in \mathfrak{D}_{\text{f}}[\alpha_{p[\ell_0 \cdot \ell_1]}(\mathcal{E})]\} \in \mathfrak{Dep}_{\text{t}}$$

In the following we assume we are able to compute an over-approximation of $\mathfrak{D}_{\text{loc}}$ and write $\mathfrak{D}^{\text{a}}_{\text{loc}}$ for this approximation.

Approximation of the dependences along a path: We need to set up a counterpart for $\boxdot$ on $\mathfrak{Dep}_{\text{t}}$; which should approximate the concatenation of traces.

Definition 8.2.10. Approximation for composition.

We let the $\boxast$ operator be defined on $\mathfrak{Dep}_{\text{t}}$ by:

$$\begin{array}{rl} \forall \mathfrak{D}_0, \mathfrak{D}_1 \in \mathfrak{Dep}_{\text{t}}, & \\ \mathfrak{D}_0 \boxast \mathfrak{D}_1 = & \{((\ell_0, x_0), (\ell_2, x_2)) \mid \\ & \quad \exists l_1 \in \mathbb{L}, x_1 \in \mathbb{X},\ ((\ell_0, x_0), (\ell_1, x_1)) \in \mathfrak{D}_0 \wedge ((\ell_1, x_1), (\ell_2, x_2)) \in \mathfrak{D}_1\} \end{array}$$

Lemma 8.2.4. Algebraic properties of $\boxast$.

The operator $\boxast$ enjoys the following properties:

1. *it is* monotone*: if $\mathfrak{D}_0, \mathfrak{D}'_0, \mathfrak{D}_1, \mathfrak{D}'_1 \in \mathfrak{Dep}_{\text{t}}$ are such that $\mathfrak{D}_0 \subseteq \mathfrak{D}'_0$ and $\mathfrak{D}_1 \subseteq \mathfrak{D}'_1$, then $\mathfrak{D}_0 \boxast \mathfrak{D}_1 \subseteq \mathfrak{D}'_0 \boxdot \mathfrak{D}'_1$.*
2. *it is* distributive over $\cup$*:*

$$\forall \mathfrak{D}_0, \mathfrak{D}'_0, \mathfrak{D}_1 \in \mathfrak{Dep}_{\text{t}}, \left\{ \begin{array}{rcl} (\mathfrak{D}_0 \cup \mathfrak{D}'_0) \boxast \mathfrak{D}_1 & = & (\mathfrak{D}_0 \boxast \mathfrak{D}_1) \cup (\mathfrak{D}'_0 \boxast \mathfrak{D}_1) \\ \mathfrak{D}_1 \boxast (\mathfrak{D}_0 \cup \mathfrak{D}'_0) & = & (\mathfrak{D}_1 \boxast \mathfrak{D}_0) \cup (\mathfrak{D}_1 \boxast \mathfrak{D}'_1) \end{array} \right.$$

3. *it is* associative*:*

$$\forall \mathfrak{D}_0, \mathfrak{D}_1, \mathfrak{D}_2 \in \mathfrak{Dep}_{\text{t}},\ \mathfrak{D}_0 \boxast (\mathfrak{D}_1 \boxast \mathfrak{D}_2) = (\mathfrak{D}_0 \boxast \mathfrak{D}_1) \boxast \mathfrak{D}_2$$

Proof.

Straightforward algebraic proofs. □

Many definitions for dependence analyses and security analyses involve a type-system. In fact, such type-system-based analyses hide a fixpoint definition [Cou97b], and we wish to make the fixpoint explicit, so as to be able to perform various refinements, such as using a better iteration strategy, computing a reduced product analysis, augmenting the control states with partitioning tokens...

Semantics as a *strongly closed* set of traces: We recall that we are using the *strongly closed* version of the semantics of programs in this chapter.

Computable approximation: We intend to prove the correctness of the approximation of the dependences of a program with a least-fixpoint equation, defined as follows:

Theorem 8.2.5. Approximation of dependences.

We assume that $\mathcal{E}$ is a strongly closed set of traces; $\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}$ is a sound approximation of the local dependences in $\mathcal{E}$. We let the backward dependence analysis function $F_{\overleftarrow{\mathfrak{D}}}$ *and $\Delta_{\mathfrak{D}}$ be defined by:*

$$\begin{array}{rrcl} F_{\overleftarrow{\mathfrak{D}}}: & \mathfrak{Dep}_{\mathrm{t}} & \rightarrow & \mathfrak{Dep}_{\mathrm{t}} \\ & D & \mapsto & D \cup \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \circledast D \end{array}$$
$$\Delta_{\mathfrak{D}} = \{((l, x), (l, x)) \mid l \in \mathbb{L}, x \in \mathbb{X}\} \in \mathfrak{Dep}_{\mathrm{t}}$$

Then,

$$\mathfrak{D}_t[\mathcal{E}] \subseteq \mathbf{lfp}_{\Delta_{\mathfrak{D}}} F_{\overleftarrow{\mathfrak{D}}} = \bigcup_{n \in \mathbb{N}} F^n_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})$$

First, we prove that $F_{\overleftarrow{\mathfrak{D}}}$ computes over-approximations for the dependences along paths.

Lemma 8.2.6. Path Composition.

Let $l_\vdash, l_\dashv \in \mathbb{L}$, $p \in \mathcal{P}(l_\vdash, l_\dashv)$, and $n = \mathbf{len}(p)$. Then,

$$\forall (x, x') \in \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p],\ ((l_\vdash, x), (l_\dashv, x')) \in F^n_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})$$

Proof.

We prove this property by induction on the length n of p:

- if $n = 0$, then p writes down $p = l_0$; hence, $\alpha_{p[p]}(\mathcal{E}) = \{\langle (l_0, \rho_0) \rangle \mid \rho_0 \in \mathbb{M}\}$, $\alpha_{p\mathcal{F}[p]}(\mathcal{E}) = \lambda(\rho \in \mathbb{M}).\{\rho\}$ and $\mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p] = \{(x_0, x_0) \mid x_0 \in \mathbb{X}\}$. Therefore, if $(x, x') \in \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p]$, then $x = x'$ and $((l_0, x), (l_0, x')) \in \Delta_{\mathfrak{D}} = F^0_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})$.
- if $n \geq 0$, then, we assume that the property holds for any path of length n and prove it for a path p of length $n + 1$.
  We assume that $p = l_0 \cdot l_1 \cdot \ldots \cdot l_n \cdot l_{n+1}$. We write $p' = l_0 \cdot l_1$ and $p'' = l_1 \cdot l_{n+1}$ (ie. $l_\vdash = l_0$ and $l_\dashv = l_{n+1}$). Then:

$$\begin{array}{lll} \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p] & & \\ = & \mathfrak{D}_{\mathrm{f}}[\alpha_{p\mathcal{F}[p]}(\mathcal{E})] & \\ = & \mathfrak{D}_{\mathrm{f}}[\alpha_{p\mathcal{F}[p'']}(\mathcal{E}) \circ \alpha_{p\mathcal{F}[p']}(\mathcal{E})] & \text{by lemma 3.2.2 and strong closure} \\ \subseteq & \mathfrak{D}_{\mathrm{f}}[\alpha_{p\mathcal{F}[p']}(\mathcal{E})] \boxdot \mathfrak{D}_{\mathrm{f}}[\alpha_{p\mathcal{F}[p'']}(\mathcal{E})] & \text{by theorem 8.2.2} \\ = & \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p'] \boxdot \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p''] & \end{array}$$

Note that the closure of $\mathcal{E}$ would not be enough: it would give the inclusion $\alpha_{p_{\mathcal{F}}[p]}(\mathcal{E}) \subseteq \alpha_{p_{\mathcal{F}}[p'']}(\mathcal{E}) \circ \alpha_{p_{\mathcal{F}}[p']}(\mathcal{E})$ but $\mathfrak{D}_{\mathrm{f}}[.]$ *is not monotone*, as we pointed out in Remark 8.2.1.
Let $(x_0, x_{n+1}) \in \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p]$. We draw from the above inequality that there exists $x_1 \in \mathbb{X}$ such that $(x_0, x_1) \in \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p']$ and $(x_1, x_{n+1}) \in \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p'']$. As a consequence:

- $(x_0, x_1) \in \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p']$ and p' is a path of length 1, so $((\ell_0, x_0), (\ell_1, x_1)) \in \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}$;
- $(x_1, x_{n+1}) \in \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p'']$, and p'' has length n; therefore, we can apply the induction hypothesis to p''; we deduce that $((\ell_1, x_1), (\ell_{n+1}, x_{n+1})) \in F^n_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})$.

Hence,

$$\begin{array}{rcl}
((\ell_0, x_0), (\ell_{n+1}, x_{n+1})) & \in & \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^n_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \\
((\ell_0, x_0), (\ell_{n+1}, x_{n+1})) & \in & \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^n_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cup F^n_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \\
((\ell_0, x_0), (\ell_{n+1}, x_{n+1})) & \in & F^{n+1}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})
\end{array}$$

As a consequence, $\forall (x_0, x_{n+1}) \in \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p],\ ((\ell_0, x_0), (\ell_{n+1}, x_{n+1})) \in F^{n+1}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})$.
This concludes the proof of the lemma. □

We now come back to the proof of the main theorem:
Proof.

We have two subproofs to complete:

- **Definition of the least-fixpoint:** Let us note that $F_{\overleftarrow{\mathfrak{D}}}$ is continuous; hence, the least-fixpoint is defined.
- **Soundness:** Let $((\ell_0, x_0), (\ell_1, x_1)) \in \mathfrak{D}_t[\mathcal{E}]$. So, $(x_0, x_1) \in \mathfrak{D}_f[\mathcal{E} \mid \ell_0, \ell_1]$. Consequently, we deduce from Theorem 8.2.3 that there exists some path $p \in \mathcal{P}(\ell_0, \ell_1)$ such that $(x_0, x_1) \in \mathfrak{D}_{\mathcal{F}}[\mathcal{E} \mid p]$. If we write $n = \mathbf{len}(p)$, then by application of lemma 8.2.6, $((\ell_0, x_0), (\ell_1, x_1)) \in F^n_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})$. The conclusion is: $((\ell_0, x_0), (\ell_1, x_1)) \in \mathbf{lfp}_{\Delta_{\mathfrak{D}}} F_{\overleftarrow{\mathfrak{D}}}$.

As a conclusion, $\mathbf{lfp}_{\Delta_{\mathfrak{D}}} F_{\overleftarrow{\mathfrak{D}}}$ exists and $\mathfrak{D}_t[\mathcal{E}] \subseteq \mathbf{lfp}_{\Delta_{\mathfrak{D}}} F_{\overleftarrow{\mathfrak{D}}}$. □

8.2.4 Dependence analysis

Theorem 8.2.5 provides a very useful approximation for the dependences of a transition system, expressed as a least fixpoint. However, a few points should still be addressed before an efficient and usable dependence analysis can be implemented:

- effective definition of $\mathfrak{D}_{\mathrm{loc}}$;
- computability of the least fixpoint;
- refinement of the analysis.

Approximation of local dependences: First, we focus on the definition of an approximation $\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}$ for $\mathfrak{D}_{\mathrm{loc}}$. Theorem 8.2.8 gives a straightforward approximation for the dependences induced by a symbolic transfer function, which we will refine later. This approximation corresponds to the syntactic approximation commonly used e.g., in slicing. Before, we prove the main theorem, we mention that the value of an expression e depends at most on the variables in e:

Definition 8.2.11. Used variables.

The set $\mathbf{use}(e)$ *of variables used in an expression* $e \in \mathbb{e}$ *is defined by a straightforward induction over* e*:*

$$\begin{array}{lrcl} \forall v \in \mathbb{V}, & \mathbf{use}(v) & = & \emptyset \\ \forall x \in \mathbb{X}, & \mathbf{use}(x) & = & \{x\} \\ \forall e_0, e_1 \in \mathbb{e}, & \mathbf{use}(e_0 \oplus e_1) & = & \mathbf{use}(e_0) \cup \mathbf{use}(e_1) \end{array}$$

Lemma 8.2.7. Dependence of an expression.

Let $e \in \mathbb{e}, \rho \in \mathbb{M}, x \in \mathbb{X}, v, v' \in \mathbb{V}$*. Then,*

$$[\![e]\!](\rho[x \leftarrow v]) \neq [\![e]\!](\rho[x \leftarrow v']) \Longrightarrow x \in \mathbf{use}(e)$$

Proof.

Straightforward induction on the structure of e □

Theorem 8.2.8. Dependence of a symbolic transfer function.

Let $\delta \in \mathbb{S}$*. The dependence* $\mathfrak{D}_{\mathrm{f}}[[\![\delta]\!]]$ *(*$\mathfrak{D}_{\mathrm{f}}[\delta]$ *for short) can be approximated by* $\mathfrak{D}^a_f[\delta]$*, which is computed by induction over* δ *as follows:*

$$\begin{array}{rcl} \mathfrak{D}^a_f[\Box] & = & \emptyset \\ \mathfrak{D}^a_f[\lfloor x_0 \leftarrow e_0, \ldots, x_n \leftarrow e_n \rfloor] & = & \{(x, x_i) \mid x \in \mathbf{use}(e_i)\} \cup \{(x, x) \mid \forall i,\ x \neq x_i\} \\ \mathfrak{D}^a_f[\lfloor e \ ?\ \delta_t \mid \delta_f \ \rfloor] & = & \{(x, y) \mid x \in \mathbf{use}(e), y \in \mathbb{X}\} \cup \mathfrak{D}^a_f[\delta_t] \cup \mathfrak{D}^a_f[\delta_f] \end{array}$$

Proof.

By induction on the structure of δ:

- case of $\Box$:
  Let $x_0, x_1 \in \mathbb{X}$, $\rho \in \mathbb{M}$, and $v, v' \in \mathbb{V}$. Then, $[\![\Box]\!](\rho[x_0 \leftarrow v])(x_1) = \emptyset = [\![\Box]\!](\rho[x_0 \leftarrow v'])(y_1)$, so $(x_0, x_1) \notin \mathfrak{D}_{\mathrm{f}}[\Box]$. Hence, $\mathfrak{D}_{\mathrm{f}}[\Box] = \emptyset$.
- case of $\delta = \lfloor x_0 \leftarrow e_0, \ldots, x_n \leftarrow e_n \rfloor$:
  Let $(y_0, y_1) \in \mathfrak{D}_{\mathrm{f}}[\Box]$, $\rho \in \mathbb{M}$, $v, v' \in \mathbb{V}$ such that $[\![\delta]\!](\rho[y_0 \leftarrow v])(y_1) \neq [\![\delta]\!](\rho[y_0 \leftarrow v'])(y_1)$. There are two cases:
  - if $y_1 = x_i$: Then, $[\![\delta]\!](\rho[y_0 \leftarrow v])(y_1) = [\![e_i]\!](\rho[y_0 \leftarrow v])$ and $[\![\delta]\!](\rho[y_0 \leftarrow v'])(y_1) = [\![e_i]\!](\rho[y_0 \leftarrow v'])$; so $[\![e_i]\!](\rho[y_0 \leftarrow v]) \neq [\![e_i]\!](\rho[y_0 \leftarrow v'])$; hence, $y_0 \in \mathbf{use}(e_i)$.
  - if $\forall i,\ y_1 \neq x_i$: Then, $[\![\delta]\!](\rho[y_0 \leftarrow v]) = \rho[y_0 \leftarrow v](y_1)$ and $[\![\delta]\!](\rho[y_0 \leftarrow v')](y_1) = \rho[y_0 \leftarrow v'](y_1)$; hence, $[\![\delta]\!](\rho[y_0 \leftarrow v])(y_1) \neq [\![\delta]\!](\rho[y_0 \leftarrow v'])(y_1)$ entails that $y_0 = y_1$.

As a conclusion, $\mathfrak{D}_{\mathrm{f}}[\delta] \subseteq \{(x, x_i) \mid x \in \mathbf{use}(e_i)\} \cup \{(x, x) \mid \forall i,\ x \neq x_i\}$.

- case of $\delta = \lfloor e \ ?\ \delta_t \mid \delta_f \ \rfloor$:
  Let $(y_0, y_1) \in \mathfrak{D}_{\mathrm{f}}[\Box]$, $\rho \in \mathbb{M}$, $v, v' \in \mathbb{V}$ such that $[\![\delta]\!](\rho[y_0 \leftarrow v])(y_1) \neq [\![\delta]\!](\rho[y_0 \leftarrow v'])(y_1)$. There are two cases:
  - if $[\![e]\!](\rho[y_0 \leftarrow v]) = [\![e]\!](\rho[y_0 \leftarrow v'])$: then, either δ_t, or δ_f, or δ_t and δ_f are executed in both cases, so it follows that $(y_0, y_1) \in \mathfrak{D}_{\mathrm{f}}[\delta_t] \cup \mathfrak{D}_{\mathrm{f}}[\delta_f]$.
  - if $[\![e]\!](\rho[y_0 \leftarrow v]) \neq [\![e]\!](\rho[y_0 \leftarrow v'])$ then, $y_0 \in \mathbf{use}(e)$.

  Therefore, $\mathfrak{D}_{\mathrm{f}}[\delta] \subseteq \{(x, y) \mid x \in \mathbf{use}(e), y \in \mathbb{X}\} \cup \mathfrak{D}_{\mathrm{f}}[\delta_t] \cup \mathfrak{D}_{\mathrm{f}}[\delta_f]$. We apply the induction hypothesis and draw the conclusion that $\mathfrak{D}_{\mathrm{f}}[\delta] \subseteq \{(x, y) \mid x \in \mathbf{use}(e), y \in \mathbb{X}\} \cup \mathfrak{D}^a_f[\delta_t] \cup \mathfrak{D}^a_f[\delta_f]$.

As a consequence, $\forall \delta \in \mathbb{\delta},\ \mathfrak{D}_{\mathrm{f}}[\delta] \subseteq \mathfrak{D}^a_f[\delta]$. $\Box$

Remark 8.2.2. Dependences and aliases.

Let us assume we consider a language which features aliasing, and that x and y point to the same memory location. Obviously, if z depends on $\star x$, then it depends also on $\star y$. Therefore, in presence of aliasing, we would have to perform some kind of alias analysis [CBC93, Deu94] first, and then use the results so as to compute the local dependences.

In the following sections, we will introduce many refinements for this approximation of $\mathfrak{D}_{\mathrm{f}}[\delta]$. In particular, the restriction of the inputs/outputs of a function (e.g., due to semantic slicing) may remove dependences.

Another significant improvement in precision comes from the ability to compose symbolic transfer functions and compute dependences globally for a path, instead of composing several approximation. We already pointed out in Section 3.2.6 that the global approximation of paths may improve the precision of static analysis. The following example demonstrate this phenomenon in dependence analysis.

Definition 8.2.4. Precision improvement.

Let us consider the following transfer functions:

$$\delta_0 = \lfloor x \ ?\ \iota \mid \Box \ \rfloor \qquad \delta_1 = \lfloor z \leftarrow x \vee y \rfloor$$

Then, $\mathfrak{D}^a_f[\delta_0] = \{(u, u) \mid u \in \mathbb{X}\} \cup \{(x, u) \mid u \in \mathbb{X}\}$ *and* $\mathfrak{D}_{\mathrm{f}}[\delta_1] = \{(u, u) \mid u \in \mathbb{X}, u \neq z\} \cup \{(x, z), (y, z)\}$*; so:*

$$\begin{array}{rl}
\multicolumn{2}{l}{\mathfrak{D}_{\mathrm{f}}[\delta_0] \boxdot \mathfrak{D}_{\mathrm{f}}[\delta_0]} \\
= & \{(u, u) \mid u \in \mathbb{X}\} \boxdot \{(u, u) \mid u \in \mathbb{X}, u \neq z\} \\
& \cup \{(x, u) \mid u \in \mathbb{X}\} \boxdot \{(u, u) \mid u \in \mathbb{X}, u \neq z\} \\
& \cup \{(u, u) \mid u \in \mathbb{X}\} \boxdot \{(x, z), (y, z)\} \\
& \cup \{(x, u) \mid u \in \mathbb{X}\} \boxdot \{(x, z), (y, z)\} \\
= & \{(u, u) \mid u \in \mathbb{X}, u \neq z\} \cup \{(x, u) \mid u \in \mathbb{X}, u \neq z\} \cup \{(x, z), (y, z)\} \cup \{(x, z)\} \\
= & \{(x, u) \mid u \in \mathbb{X}\} \cup \{(u, u) \mid u \in \mathbb{X}, u \neq z\} \cup \{(y, z)\}
\end{array}$$

However, $\mathit{simplify}(\delta_1 \oplus \delta_0) = \mathit{simplify}(\lfloor x \ ?\ \lfloor z \leftarrow x \vee y \rfloor \mid \Box \ \rfloor) = \lfloor x \ ?\ \lfloor z \leftarrow \mathbf{true} \rfloor \mid \Box \ \rfloor$*, so that* $\mathfrak{D}_{\mathrm{f}}[\mathit{simplify}(\delta_1 \oplus \delta_0)] = \{(x, u) \mid u \in \mathbb{X}\} \cup \{(u, u) \mid u \in \mathbb{X}, u \neq z\}$.

We gave an encoding of all the one-step transitions in symbolic transfer functions in Figure 3.5; therefore, an approximation for $\mathfrak{D}_{\text{loc}}$ follows from the approximation of the dependences induced by any transfer function (Theorem 8.2.8).

Computability: Theorem 8.2.5 provides a least-fixpoint approximation for the dependences induced by a set of traces, hence by a program. We mentioned that function $F_{\overleftarrow{\mathfrak{D}}}$ is continuous, so the least-fixpoint is reached after ω iterations: $\mathbf{lfp}_{\Delta_{\mathfrak{D}}} F_{\overleftarrow{\mathfrak{D}}} = \cup\{F^n_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \mid n \in \mathbb{N}\}$.

However, $\mathfrak{Dep}_{\text{t}}$ is finite, since the number of control states in a program is finite and so is the number of variables. Therefore, the least-fixpoint can in fact be reached after a finite number of iterations.

Remark 8.2.3. Procedural programs.

If we consider a procedural analysis, then the calling stack is part of the control states. If a program contains recursive functions, then the set of control states is no longer finite, since there exist an infinity of control stacks; then, we should apply some abstractions to the stacks, as in Section 4.1.1 (such abstractions can be defined as extended systems, as in Section 4.2).
Similarly, in case dynamic memory allocation is allowed, then the number of possible memory cells is infinite, so an abstraction for memory locations should be defined.

Improving precision: We pointed out in Example 8.2.3 some imprecision inherent in the approximation of the dependences between two points with the join of the dependences along all paths between these two points (Theorem 8.2.3).

Let $l_{\vdash}, l_{\dashv} \in \mathbb{L}$. Intuitively, if a variable x is not modified on any path between $l_{\vdash}$ and $l_{\dashv}$ if $l_{\vdash}$ precedes $l_{\dashv}$ (i.e., any execution reaching $l_{\vdash}$ eventually reaches $l_{\dashv}$), then x at $l_{\dashv}$ may not depend on any variable but x at $l_{\vdash}$. Let us formalize this argument:

Definition 8.2.12. Control state precedence.

Let $l_{\vdash}, l_{\dashv} \in \mathbb{L}$. We say that $l_{\vdash}$ precedes *$l_{\dashv}$ (implicitly: with respect to a set of traces $\mathcal{E}$, or with the semantics of program s), which we denote by $l_{\vdash} \prec l_{\dashv}$ if and only if:*

$$\forall \rho \in \mathbb{M},\ \exists \langle s_0, \ldots, s_n \rangle \in \mathcal{E},\ s_0 = (l, \rho) \wedge \exists \rho' \in \mathbb{M} s_n = (l', \rho')$$

The relation $\prec$ is transitive; it is generally neither reflexive (case of unreachable states) nor antisymmetric (case of e.g., loops).

Then, the criterion evoked above can be stated as follows:

Theorem 8.2.9. Precedence, dependence and variable update.

Let $l, l' \in \mathbb{L}$ such that $l \prec l'$, and $x, x' \in \mathbb{X}$ such that $((l, x), (l', x')) \in \mathfrak{D}_t[\mathcal{E}]$, and $x \neq x'$. Then, there exists a path from l to l' where the value of x' changes, ie. is *updated at least once; in fact the following stronger result holds:*

$$\exists \rho \in \mathbb{M},\ \exists p \in \mathcal{P}(\iota, \iota'),\ \exists v \in \alpha_{p\mathcal{F}[p]}(\mathcal{E})(\rho)(x') \wedge v \notin \rho(x')$$

Proof.

Let us assume that $\iota \prec \iota'$, $((\iota, x), (\iota', x')) \in \mathfrak{D}_t[\mathcal{E}]$, and $x \neq x'$. There exist $\rho \in \mathbb{M}, v_0, v_1 \in \mathbb{V}$, such that $\phi(\rho_0)(x') \neq \phi(\rho_1)(x')$, where $\phi = \alpha_{p\mathcal{F}[\iota]}\iota'(\mathcal{E})$, and $\forall i \in \{0,1\},\ \rho_i = \rho[x \leftarrow v_i]$.
The precedence property entails that $\forall i \in \{0,1\},\ \phi(\rho_i)(x') \neq \emptyset$. The dependence entails that $\exists i,\ \phi(\rho_i)(x') \neq \{\rho(x')\}$; hence, $\exists v \in \mathbb{V},\ v \in \phi(\rho_i)(x')$. The main result follows. □

The precedence relation can be computed syntactically; moreover, the set of variables modified between any pair of control states can be approximated by a simple static analysis, which we do not describe here. More precisely, this analysis would over-approximate the set of tuples (ι, ι', x') such that x' is modified on at least one path from ι to ι'.

Then, the dependence analysis is the result of a reduced product of the analysis described in Theorem 8.2.5 and of the analysis approximating the variable updates.

Definition 8.2.5. Precedence among control states.
Let us consider the definition of the $\prec$ relation in the case of the simple language introduced in Section 2.2:

- *if P contains an assignment $\iota_0 : x := e; \iota_1$, then, clearly $\iota_0 \prec \iota_1$ (the case of input statements, sequences,* **if** *statements are similar);*
- *the case of an assert statements ι_0 :* **assert**$(e); \iota_1$ *is more interesting:*
  - *in the standard settings, some traces from ι_0 may not reach ι_1, due to the assertion being violated; as a consequence $\iota_0 \not\prec \iota_1$ (so we keep the spurious dependences mentioned in the end of Section 8.2.2);*
  - *by contrast, in the "lazy semantics" approach [CF89], then any trace from ι_0 eventually reaches ι_1 (since an erroneous trace continues, with an error flag enabled), so that $\iota_0 \prec \iota_1$.*

The case of a loop statement ι_0 : **while**$(e)\{\ldots\}; \iota_1$ *is similar (i.e., $\iota_0 \prec \iota_1$ holds only in the lazy semantics approach). As a consequence, we confirm that our framework accommodates both approaches; in practice though, we use the lazy one (since we are interested in backward dependences from errors only).*

Definition 8.2.6. Precedences among control states (Example 8.2.3 continued).
Clearly, $\iota_0 \prec \iota_5$; moreover, if $y \in \mathbb{X} \setminus \{x\}$, then y is not modified on any path between ι_0 and ι_5, so there is no dependence $(\iota_5, y) \leadsto (\iota_0, b)$ (in fact, $(\iota_5, y) \leadsto (\iota_0, z) \Longrightarrow z = y$).

In practice, most implementations of dependence analyses distinguish data and control dependences, which avoid the need for this simple refinement. However, the advantage of our approach is to start with a semantic definition of dependences, to derive a rather rough computable approximation and to refine it later. The price to pay for this approach was the need to recover the distinction between data and control dependences.

8.2.5 Dependence graphs

Backward dependence: Theorem 8.2.5 provides a means to compute *all* the dependences in a program. However, one usually does not need to compute all the dependences: the purpose of the dependence analysis is usually to figure out what may affect the value of a variable x at point ℓ, or to extract a slice.

As a consequence, we define the notion of criterion, which states what part of the program we wish to compute the dependences of:

Definition 8.2.13. Criterion.
A criterion $\mathcal{C}$ is a set of pairs made of a control point and a variable $\mathcal{C} \in \mathcal{P}(\mathbb{L} \times \mathbb{X})$.

In particular, if an alarm is raised at point ℓ due to the possible failure of an assertion **assert**(e);, then, we should consider the criterion $\mathcal{C} = \{\ell\} \times \mathbf{use}(e)$.

The set of entities the criterion depends on is defined as follows:

Definition 8.2.14. Backward dependence induced by a criterion.
Let $\mathcal{E}$ be a strongly closed set of traces, and $\mathcal{C} \in \mathcal{P}(\mathbb{L} \times \mathbb{X})$. Then, the backward dependence *induced by $(\mathcal{E}, c)$ is the set of dependences $\overleftarrow{\mathfrak{dep}}[\mathcal{E}](\mathcal{C})$ defined by:*

$$\overleftarrow{\mathfrak{dep}}[\mathcal{E}](\mathcal{C}) = \{((l, x), (l', x')) \in \mathfrak{D}_t[\mathcal{E}] \mid (l', x') \in \mathcal{C}\} = \mathfrak{D}_t[\mathcal{E}] \cap \overleftarrow{\mathcal{C}}_\pi$$

where $\overleftarrow{\mathcal{C}}_\pi = (\mathbb{L} \times \mathbb{X}) \times \mathcal{C}$.

Extraction of a backward dependence: We propose to derive an algorithm for extracting the backward dependence induced by a criterion from the fixpoint-based definition of all the dependences of a program.

In the following, we write $\Delta_{\mathfrak{D}}^{\mathcal{C}}$ for $\mathcal{C}^2$.

Theorem 8.2.10. Backward dependence analysis.
We let $\overleftarrow{\mathfrak{dep}}^{\mathrm{a}}[\mathcal{E}](\mathcal{C}) = \mathbf{lfp}_{\Delta_{\mathfrak{D}}^{\mathcal{C}}} F_{\overleftarrow{\mathfrak{D}}}$.
The backward dependence can be safely approximated by $\overleftarrow{\mathfrak{dep}}^{\mathrm{a}}[\mathcal{E}](\mathcal{C})$:

$$\overleftarrow{\mathfrak{dep}}[\mathcal{E}](\mathcal{C}) \subseteq \overleftarrow{\mathfrak{dep}}^{\mathrm{a}}[\mathcal{E}](\mathcal{C})$$

Proof.

First, we prove that $F_{\overleftarrow{\mathfrak{D}}}^{n}(\Delta_{\mathfrak{D}}) \cap \overleftarrow{\mathcal{C}}_\pi = F_{\overleftarrow{\mathfrak{D}}}^{n}(\Delta_{\mathfrak{D}}^{\mathcal{C}})$ by induction over n:

- if $n = 0$, then: $F^0_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cap \overleftarrow{\mathcal{C}}_\pi = \Delta_{\mathfrak{D}} \cap \overleftarrow{\mathcal{C}}_\pi = \mathcal{C}^2 = \Delta^{\mathcal{C}}_{\mathfrak{D}} = F^0_{\overleftarrow{\mathfrak{D}}}(\Delta^{\mathcal{C}}_{\mathfrak{D}})$.
- if $n \geq 0$, let us assume the property holds for n and show it for $n+1$:

$$\begin{array}{rcl} F^{n+1}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cap \overleftarrow{\mathcal{C}}_\pi & = & \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast \left(F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cup F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})\right) \cap \overleftarrow{\mathcal{C}}_\pi \\ & = & \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast \left(F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})\right) \cap \overleftarrow{\mathcal{C}}_\pi \cup \left(F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cap \overleftarrow{\mathcal{C}}_\pi\right) \end{array}$$

Let $\ell, \ell'' \in \mathbb{L}, x, x'' \in \mathbb{X}$ and let us consider the first term:

$$\begin{array}{l} ((\ell, x), (\ell'', x'')) \in \left(\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})\right) \cap \overleftarrow{\mathcal{C}}_\pi \\ \iff \left\{\begin{array}{l} ((\ell, x), (\ell'', x'')) \in \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \\ ((\ell, x), (\ell'', x'')) \in \overleftarrow{\mathcal{C}}_\pi \end{array}\right. \\ \iff \exists \ell' \in \mathbb{L}, x' \in \mathbb{X}, \left\{\begin{array}{l} ((\ell, x), (\ell', x')) \in \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \\ ((\ell', x'), (\ell'', x'')) \in F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \\ (\ell'', x'') \in \mathcal{C} \end{array}\right. \\ \iff \exists \ell' \in \mathbb{L}, x' \in \mathbb{X}, \left\{\begin{array}{l} ((\ell, x), (\ell', x')) \in \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \\ ((\ell', x'), (\ell'', x'')) \in F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cap \overleftarrow{\mathcal{C}}_\pi \end{array}\right. \\ \iff ((\ell, x), (\ell'', x'')) \in \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast \left(F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cap \overleftarrow{\mathcal{C}}_\pi\right) \end{array}$$

As a consequence,

$$\begin{array}{lll} F^{n+1}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cap \overleftarrow{\mathcal{C}}_\pi & & \\ = \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast \left(F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cap \overleftarrow{\mathcal{C}}_\pi\right) \cup \left(F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cap \overleftarrow{\mathcal{C}}_\pi\right) & & \\ = \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta^{\mathcal{C}}_{\mathfrak{D}}) \cup F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta^{\mathcal{C}}_{\mathfrak{D}}) & & \text{(induction hypothesis)} \\ = F^{n+1}_{\overleftarrow{\mathfrak{D}}}(\Delta^{\mathcal{C}}_{\mathfrak{D}}) & & \end{array}$$

The intermediate result follows.
From this point, the proof of the theorem is straightforward. Indeed:

$$\begin{array}{lll} \overleftarrow{\mathfrak{dep}}[\mathcal{E}](\mathcal{C}) & & \\ = \mathfrak{D}_t[\mathcal{E}] \cap \overleftarrow{\mathcal{C}}_\pi & & \\ \subseteq (\mathbf{lfp}_{\Delta_{\mathfrak{D}}} F_{\overleftarrow{\mathfrak{D}}}) \cap \overleftarrow{\mathcal{C}}_\pi & & \text{by Theorem 8.2.5} \\ = \left(\bigcup_{n \in \mathbb{N}} F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})\right) \cap \overleftarrow{\mathcal{C}}_\pi & & \\ = \bigcup_{n \in \mathbb{N}} \left(F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cap \overleftarrow{\mathcal{C}}_\pi\right) & & \\ = \bigcup_{n \in \mathbb{N}} F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta^{\mathcal{C}}_{\mathfrak{D}}) & & \text{due to the intermediate result} \\ = \mathbf{lfp}_{\Delta^{\mathcal{C}}_{\mathfrak{D}}} F_{\overleftarrow{\mathfrak{D}}} & & \end{array}$$

This concludes the proof of the theorem.
□

In practice, a pre-analysis phase collects all the local dependences of the program in a *dependence graph* [HRB90]. Then, backward dependences can be extracted by computing a closure (i.e., fixpoint computation) of the dependences in the graph, starting from the criterion.

Definition 8.2.7. Backward dependence (Example 8.1.1 continued).

Let us consider the program P displayed in Figure 8.1(a) (Example 8.1.1). We focus on the backward dependence induced by the criterion (ℓ_5, y).

Then, all the local dependences involved in the computation of $\overleftarrow{\mathfrak{dep}}^{\mathrm{a}}[\llbracket P \rrbracket](\{(\ell_5, y)\})$ are displayed in Figure 8.2. As a result $\overleftarrow{\mathfrak{dep}}^{\mathrm{a}}[\llbracket P \rrbracket](\{(\ell_5, y)\})$ is equal to the set of dependences,

$$\begin{tikzcd}
 & & (\ell_3, y) \arrow[r] \arrow[ddr] & (\ell_0, y) \\
(\ell_5, y) \arrow[r] \arrow[dr] & (\ell_4, y) \arrow[ur] \arrow[r] & (\ell_3, z) \arrow[r] \arrow[dr] & (\ell_0, z) \\
 & (\ell_2, y) \arrow[r] & (\ell_1, x) \arrow[r] & (\ell_0, x)
\end{tikzcd}$$

Figure 8.2: Local dependences involved in the approximation of the backward dependence induced by $\{(\ell_5, y)\}$

$$\{(\ell_5, y), (\ell_4, y), (\ell_3, y), (\ell_3, z), (\ell_2, y), (\ell_1, x), (\ell_0, x), (\ell_0, y), (\ell_0, z)\} \times \{(\ell_5, y)\}$$

Forward dependences: The fixpoint algorithms proposed in Theorem 8.2.5 and Theorem 8.2.10 work *backwards*: they seek for dependences in the opposite direction compared to the execution paths. We could propose *forward* algorithms as well.

In particular, we let the *forward dependence semantic function* be defined by:

$$\begin{array}{rccl} F_{\overrightarrow{\mathfrak{D}}} : & \mathfrak{Dep}_{\mathrm{t}} & \rightarrow & \mathfrak{Dep}_{\mathrm{t}} \\ & D & \mapsto & D \cup D \circledast \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \end{array}$$

The forward analysis provides the same approximation of the dependences of a set of traces:

Theorem 8.2.11. Forward approximation of dependences.

Let $\mathcal{E}$ be a strongly closed set of traces. Then, $\mathbf{lfp}_{\Delta_{\mathfrak{D}}} F_{\overrightarrow{\mathfrak{D}}} = \mathbf{lfp}_{\Delta_{\mathfrak{D}}} F_{\overleftarrow{\mathfrak{D}}}$. As a consequence,

$$\mathfrak{D}_t[\mathcal{E}] \subseteq \mathbf{lfp}_{\Delta_{\mathfrak{D}}} F_{\overrightarrow{\mathfrak{D}}}$$

Proof.

This result follows from the fact that the iterates in both fixpoints are equal: $\forall n \in \mathbb{N},\ F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) = F^{n}_{\overrightarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})$. This equality can be proved by induction over n.
But, we prove first the following, by induction over n: $\forall n \in \mathbb{N},\ F^{n}_{\overleftarrow{\mathfrak{D}}}(\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}) \boxast \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} = \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^{n}_{\overleftarrow{\mathfrak{D}}}(\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}})$.

- if $n = 0$, then $F^{0}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \boxast \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} = \Delta_{\mathfrak{D}} \boxast \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} = \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} = \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast \Delta_{\mathfrak{D}} = \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^{0}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})$;
- if $n \in \mathbb{N}$, and the property holds for n, then:

$$\begin{array}{lll}
& F^{n+1}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \boxast \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} & \\
= & (\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \cup F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})) \boxast \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} & \\
= & (\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})) \boxast \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \cup F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \boxast \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} & \text{(distributivity } \cup \text{ over } \boxast) \\
= & \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast (F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \boxast \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}) \cup F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \boxast \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} & \text{(associativity of } \boxast) \\
= & \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast (\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})) \cup \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) & \text{(induction hypothesis)} \\
= & \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast (F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) \boxast \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^{n}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}})) & \text{(distributivity } \cup \text{ over } \boxast) \\
= & \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}} \boxast F^{n+1}_{\overleftarrow{\mathfrak{D}}}(\Delta_{\mathfrak{D}}) &
\end{array}$$

The proof of equality of the iterates from this point is straightforward.
Then, the equality of the forward and backward fixpoints follows from a straightforward induction on the iterates. □

Moreover, if we define the forward dependence induced by a criterion as the dual of $\overleftarrow{\mathfrak{dep}}[\mathcal{E}]$, $F_{\overrightarrow{\mathfrak{D}}}$ also provide an over-approximation for such forward dependences:

Theorem 8.2.12. Forward dependence analysis.

Let $\mathcal{E}$ be a strongly closed set of traces and $\mathcal{C} \subseteq \mathcal{P}(\mathbb{L} \times \mathbb{X})$.
The forward dependence *induced by $(\mathcal{E}, \mathcal{C})$ is the set of dependences $\overrightarrow{\mathfrak{dep}}[\mathcal{E}](\mathcal{C}) = \{((\iota, x), (\iota', x')) \in \mathfrak{D}_t[\mathcal{E}] \mid (l, x) \in \mathcal{C}\} = \mathfrak{D}_t[\mathcal{E}] \cap \overrightarrow{\mathcal{C}}_\pi$, where $\overrightarrow{\mathcal{C}}_\pi = \mathcal{C} \times (\mathbb{L} \times \mathbb{X})$.*
Then, if we let $\overrightarrow{\mathfrak{dep}}^{\mathrm{a}}[\mathcal{E}](\mathcal{C})$ by $\overrightarrow{\mathfrak{dep}}^{\mathrm{a}}[\mathcal{E}](\mathcal{C}) = \mathbf{lfp}_{\Delta^{\mathcal{C}}_{\mathfrak{D}}} F_{\overrightarrow{\mathfrak{D}}}$, then $\overrightarrow{\mathfrak{dep}}[\mathcal{E}](\mathcal{C}) \subseteq \overrightarrow{\mathfrak{dep}}^{\mathrm{a}}[\mathcal{E}](\mathcal{C})$.

Proof.

Entirely similar to Theorem 8.2.10. □

Forward dependences collect what depends on a criterion. In the following, we mostly use backward dependences, since we are interested in *causes* of results rather than in *consequences*. Hence, we usually let "dependence" mean "backward dependence", unless stated otherwise.

Most of the results and definitions given in the following sections would also apply in the case of forward (observable or abstract) dependences.

For a discussion about the applications of forward dependences (e.g., in the extraction of forward slices), we refer the reader to [HRB90, HDSS96].

8.3 Observable Dependences

We now propose a first refinement for the notion of dependences. Indeed, when considering a semantic slice, we do not consider *all* the traces of the program. As a result, we may expect to refine somewhat the algorithm for computing dependences, which we described in Section 8.2.3.

8.3.1 Dependences on semantic slices and non monotonicity

A natural approach would be to define the dependences for a semantic slice $\mathcal{E}'$ of $\mathcal{E}$ as the dependences induced by $\mathcal{E}'$. However, this definition would result in very non-intuitive dependences, which would not correspond to what we want to capture. Indeed, we recall that the $\mathcal{E} \mapsto \mathfrak{D}_t[\mathcal{E}]$ function is *not* monotone.

As a result, dependences of semantic slices would be plagued with meaningless *fictitious dependences*, as illustrated in the following example.

Definition 8.3.1. Fictitious dependences in a semantic slice.
We consider the following program s, with two variables x, y:

$$\begin{array}{ll} l_0 : & \textbf{input}(x); \\ l_1 : & \textbf{input}(y); \\ l_2 : & \ldots \end{array}$$

This program does not induce any dependence across distinct variables. However, we may consider the subset $\mathit{clos}(\mathcal{E})$ of $[\![s]\!]$, where:

$$\begin{array}{rcl} \mathcal{E} & = & \{ \quad \langle (l_0, (x=0, y=0)), (l_1, (x=4, y=0)), (l_2, (x=4, y=4)) \rangle, \\ & & \quad\;\; \langle (l_0, (x=0, y=0)), (l_1, (x=2, y=0)), (l_2, (x=2, y=2)) \rangle \quad \} \end{array}$$

Clearly, $\mathcal{E} \subseteq [\![s]\!]$. However, we note that the value read for y at l_1 is always the same as the value of x at this point; hence, this new set of traces defines a dependence of $((l_1, x), (l_2, y))$, which was not induced by $[\![s]\!]$.
This example shows the non-monotonicity of the dependence operator, even when applied to semantic slices of a same program.

We note that the dependence $(l_2, y) \rightsquigarrow (l_1, x)$ in the above example has no satisfactory interpretation. Indeed, the fact that y always has the same value of x stems from the choice of the semantic slicing criterion rather than the actual behavior of the program, even though we expect dependences to provide informations about the origin of the program results (as opposed to the semantic slicing choices).

As a consequence, we propose to work on a definition for *observable* dependences instead.

8.3.2 Observable dependences induced by a function

First, we define observable dependences of functions, with constraints on the inputs and on the outputs.

Definition 8.3.1. Function slice.

A slice *of a function $\phi \in \mathfrak{Den}$ is defined by a pair $(\mathcal{M}_i, \mathcal{M}_o) \in (\mathcal{P}(\mathbb{M}))^2$, where $\mathcal{M}_i$ is an* input constraint *and $\mathcal{M}_o$ is an* output constraint*. The meaning of this function slice is described by:*

$$\widetilde{\phi} : \rho \mapsto \begin{cases} \phi(\rho) \cap \mathcal{M}_o & \text{if } \rho \in \mathcal{M}_i \\ \emptyset & \text{if } \rho \notin \mathcal{M}_i \end{cases}$$

Observable dependences: An observable dependence is a dependence, which is revealed in the semantic slice under consideration:

Definition 8.3.2. Observable dependences.

Let $\phi \in \mathfrak{Den}$, $\mathcal{M}_i, \mathcal{M}_o \subseteq \mathbb{M}$, $x_0, x_1 \in \mathbb{X}$. We say that ϕ induces an observable dependence *of x_1 on x_0 in the semantic slice $(\mathcal{M}_i, \mathcal{M}_o)$ if and only if*

$$\exists \rho \in \mathcal{M}_i,\ \exists v_a, v_b \in \mathcal{M}_i(x_0),\ \phi(\rho[x_0 \leftarrow v_a])(x_1) \cap \mathcal{M}_o(x_1) \neq \phi(\rho[x_0 \leftarrow v_b])(x_1) \cap \mathcal{M}_o(x_1)$$

We write $x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \Mapsto \mathcal{M}_o} x_0$ if such a dependence exists. Last, we let $\mathfrak{D}_{\mathrm{sf}}[\phi; \mathcal{M}_i \Mapsto \mathcal{M}_o]$ denote the set $\{(x_0, x_1) \in \mathbb{X}^2 \mid x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \Mapsto \mathcal{M}_o} x_0\}$ of dependences observable on the semantic slice.

Definition 8.3.2. Observable dependences.

*Let us consider the input constraint $b =$ **true** and the function ϕ introduced in Example 8.2.1:*

$$\phi(\rho) = \begin{cases} \{\rho[y \leftarrow x]\} & \text{if } \rho(b) = \textbf{true} \\ \emptyset & \text{if } \rho(b) = \textbf{false} \end{cases}$$

Then, if $z \in \mathbb{X}$, we can show that z does not depend on b (it is not possible to exhibit two distinct values for b in the input constraint). As a consequence,

$$\mathfrak{D}_{\mathrm{sf}}[\phi; \mathcal{M}_i \Mapsto \mathcal{M}_o] = \{(x, y)\} \cup \{(z, z) \mid z \in \mathbb{X}\}$$

Intuitively, we focus on the executions, which satisfy the condition of an **if***-statement. As a consequence, we restrict to a single paths: all traces in the slice go through the same branch of the* **if***-statement. As a consequence, the absence of dependence on b was to be expected.*

Hierarchy of observations: The definition of observable dependences allows to recover a kind of monotonicity result:

Theorem 8.3.1. Hierarchy of observable dependences –case of functions.
Let $\mathcal{M}_i, \mathcal{M}'_i, \mathcal{M}_o, \mathcal{M}'_o \subseteq \mathbb{M}$, such that $\mathcal{M}_i \subseteq \mathcal{M}'_i$ and $\mathcal{M}_o \subseteq \mathcal{M}'_o$, and $\phi \in \mathfrak{Den}$. Then:

$$\forall x_0, x_1 \in \mathbb{X},\ x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \Mapsto \mathcal{M}_o} x_0 \Longrightarrow x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}'_i \Mapsto \mathcal{M}'_o} x_0$$

An important corollary of this property is that $\forall x_0, x_1 \in \mathbb{X},\ x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \Mapsto \mathcal{M}_o} x_0 \Longrightarrow x_1 \overset{\phi}{\rightsquigarrow}_{\mathbb{M} \Mapsto \mathbb{M}} x_0$. In other words, the observable dependences are a subset of the dependences:

$$\forall x_0, x_1 \in \mathbb{X},\ x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \Mapsto \mathcal{M}_o} x_0 \Longrightarrow x_1 \overset{\phi}{\rightsquigarrow} x_0$$

Proof.
We propose to prove two simple properties first:

- we assume that $\mathcal{M}_o = \mathcal{M}'_o$ and prove the **monotonicity with respect to the output constraint**:
  Let us assume that $x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \Mapsto \mathcal{M}_o} x_0$. Then, there exist $\rho \in \mathcal{M}_i$, $v_a, v_b \in \mathbb{V}$ such that $\forall i \in \{a, b\},\ v_i \in \mathcal{M}_i(x_0)$ and $\phi(\rho_a)(x_1) \neq \phi(\rho_b)(x_1)$, where $\rho_i = \rho[x_0 \leftarrow v_i]$. Since $\mathcal{M}_i(x_0) \subseteq \mathcal{M}'_i(x_0)$, $\forall i \in \{a, b\},\ v_i \in \mathcal{M}'_i(x_0)$, so $x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}'_i \Mapsto \mathcal{M}_o} x_0$.
- we assume that $\mathcal{M}_i = \mathcal{M}'_i$ and prove the **monotonicity with respect to the input constraint**:
  Let us assume that $x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \Mapsto \mathcal{M}_o} x_0$. Then, there exist $\rho \in \mathbb{M}$, $v_a, v_b \in \mathbb{V}$ such that $\phi(\rho_a)(x_1) \cap \mathcal{M}_o(x_1) \neq \phi(\rho_b)(x_1) \cap \mathcal{M}_o(x_1)$, where ρ_a and ρ_b are defined as usual. Then, $\mathcal{M}_o(x_1) \subseteq \mathcal{M}'_o(x_1)$, which entails $\phi(\rho_a)(x_1) \cap \mathcal{M}'_o(x_1) \neq \phi(\rho_b)(x_1) \cap \mathcal{M}'_o(x_1)$, since:

  $$\forall E, E', A, B,\ E \cap B = E' \cap B \wedge A \subseteq B \Longrightarrow E \cap A = E' \cap A$$

  As a result, $x_1 \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \Mapsto \mathcal{M}'_o} x_0$.

The result of the theorem follows from the composition of the two results above. □

Approximation of composition: The approximation of the dependences of $\phi_1 \circ \phi_0$ was a crucial step in the definition of an algorithm for approximating the dependences of a program; therefore, we extend this result here.

Theorem 8.3.2. Composition of observable dependences –approximation.
Let $\mathcal{M}_0, \mathcal{M}_1, \mathcal{M}_2 \in \mathcal{P}(\mathbb{M})$, and $\phi_0, \phi_1 \in \mathfrak{Den}$. Let $\widetilde{\phi}$ be the composition of the semantic slice $\widetilde{\phi_0}$ of ϕ_0 defined by $(\mathcal{M}_0, \mathcal{M}_1)$ with the semantic slice $\widetilde{\phi_1}$ of ϕ_1 defined by $(\mathcal{M}_1, \mathcal{M}_2)$:

$$\widetilde{\phi} : \rho \mapsto \begin{cases} \phi_1(\phi_0(\rho) \cap \mathcal{M}_1)\mathcal{M}_2 & \text{if } \rho \in \mathcal{M}_1 \\ \emptyset & \text{if } \rho \notin \mathcal{M}_0 \end{cases}$$

Then, we have the following approximation:

$$\mathfrak{D}_{\mathrm{sf}}[\widetilde{\phi}; \mathcal{M}_0 \mapsto \mathcal{M}_2] \subseteq \mathfrak{D}_{\mathrm{sf}}[\phi_0; \mathcal{M}_0 \mapsto \mathcal{M}_1] \boxdot \mathfrak{D}_{\mathrm{sf}}[\phi_1; \mathcal{M}_1 \mapsto \mathcal{M}_2]$$

Proof.

Similar to the proof of Theorem 8.2.2. □

8.3.3 Observable dependences induced by a set of traces

In this section, we consider a set of traces $\mathcal{E}$ (typically, $\mathcal{E} = [\![P]\!]$ for some program P) and a semantic slice $\mathcal{E}' \subseteq \mathcal{E}$. We propose to define the *observable* dependences, corresponding to the semantic slice $\mathcal{E}'$. Note that we assume that $\mathcal{E}'$ is strongly closed (so that the closeness of the semantic slices is addressed in the end of this subsection).

Remark 8.3.1. Strong closure of semantic slices.

The assumption that the semantic slice $\mathcal{E}'$ be strongly closed is crucial for the definition of observable dependences to make sense and also for the algorithms, which we describe in the following subsections for approximating such dependences to be sound.
As a consequence, we require that the semantic slicing function described in Chapter 7 inputs and returns strongly closed sets of traces only. In particular, we replace the definition of semantic slice (Definition 7.2.2) with the following definition ($\mathbb{C}$ is a semantic slicing domain, $c \in \mathbb{C}$):

$$\mathfrak{Slice}_{\mathbb{C}}\langle \mathcal{E}, c\rangle = clos([\![P]\!] \cap \gamma_{\mathbb{C}}(c))$$

We recall that $clos$ completes a set of traces by adding all the sub-traces, so this operator returns closed sets of traces.
Establishing the strong closure requires proving that, if $\sigma, \sigma' \in \mathfrak{Slice}_{\mathbb{C}}\langle \mathcal{E}, c\rangle$ are such that $\sigma \frown \sigma'$ is defined, then $\sigma \frown \sigma' \in \mathfrak{Slice}_{\mathbb{C}}\langle \mathcal{E}, c\rangle$. This property is trivial in the case of the initial and final states slicing criteria and of the input constraints slicing criteria.
In the case of the execution patterns criteria, the property is clearly true if we consider that the control states enclose the partitioning tokens (i.e., we should use $\mathbb{L}_{\mathbb{T}}$ instead of $\mathbb{L}$ in the dependence analysis).

Definition: The definition of observable dependences of a semantic slice extends Definition 8.3.2. The observable dependences between two points or along a path are the dependences of the underlying function, constrained with the set of input and output states which are observable in $\mathcal{E}'$, relatively to this transition.

The observable from-to dependences are defined by:

Definition 8.3.3. Observable dependences.

Let $l_0, l_1 \in \mathbb{L}$ and $x_0, x_1 \in \mathbb{X}$. We define the input and output constraints:

$$\begin{array}{rcl}\mathcal{M}_i & = & \{\rho_0 \in \mathbb{M} \mid \exists \langle (l_0, \rho_0), \ldots \rangle \in \alpha_{t[l_0,l_1]}(\mathcal{E}')\} \\ \mathcal{M}_o & = & \{\rho_1 \in \mathbb{M} \mid \exists \langle \ldots, (l_1, \rho_1) \rangle \in \alpha_{t[l_0,l_1]}(\mathcal{E}')\}\end{array}$$

We say that there exists an observable dependence of (l_1, x_1) on (l_0, x_0) and we write $(l_1, x_1) \rightsquigarrow_{[\mathcal{E}']} (l_0, x_0)$ if and only if:

$$(x_0, x_1) \in \mathfrak{D}_{\mathrm{sf}}[\alpha_{t\mathcal{F}[l_0,l_1]}(\mathcal{E}); \mathcal{M}_i \mapsto \mathcal{M}_o]$$

We write $\mathfrak{D}_{\mathrm{st}}[\mathcal{E} \mid \mathcal{E}']$ for the observable dependences induced by the semantic slice $\mathcal{E}'$ of $\mathcal{E}$; it is defined by:

$$\mathfrak{D}_{\mathrm{st}}[\mathcal{E} \mid \mathcal{E}'] = \{((l_0, x_0), (l_1, x_1)) \in (\mathbb{L} \times \mathbb{X})^2 \mid (l_1, x_1) \rightsquigarrow_{[\mathcal{E}']} (l_0, x_0)\}$$

Note that the above definition is based on the definition of constraints on the input and outputs of the function; however, it uses the function defined in the initial transition system:

The definition of observable dependences $\mathfrak{D}_{\mathrm{st}}[p \mid \mathcal{E}]\mathcal{E}'$ along a path p is similar (it is based on $\alpha_{p[p]}$ instead of $\alpha_{t[l_0,l_1]}$).

Hierarchies of observable dependences: The "monotonicity" of the observable dependences with respect to the semantic slice follows straightforwardly from Theorem 8.3.1.

Theorem 8.3.3. Hierarchy of observable dependences –case of sets of traces.

Let $\mathcal{E}_0, \mathcal{E}_1$ be two semantic slices of $\mathcal{E}$ such that $\mathcal{E}_0 \subseteq \mathcal{E}_1$. Then:

$$\mathfrak{D}_{\mathrm{st}}[\mathcal{E} \mid \mathcal{E}_0] \subseteq \mathfrak{D}_{\mathrm{st}}[\mathcal{E} \mid \mathcal{E}_1]$$

In particular, if $\mathcal{E}'$ is a semantic slice of $\mathcal{E}$, then $\mathfrak{D}_{\mathrm{st}}[\mathcal{E} \mid \mathcal{E}'] \subseteq \mathfrak{D}_{\mathrm{st}}[\mathcal{E} \mid \mathcal{E}] = \mathfrak{D}_t[\mathcal{E}]$: the dependences observable in a semantic slice form a subset of the dependences of the initial set of traces.

A very important consequence of Theorem 8.3.3 is that we can focus on the dependences of an approximation $\mathcal{E}''$ of a semantic slice $\mathcal{E}'$, when studying $\mathcal{E}'$. Indeed, we may not be able to compute $\mathcal{E}'$; hence, we would not be able to compute any safe approximation of the observable dependences of $\mathcal{E}'$ without the property proved in this Theorem.

At this point, we can illustrate the notion of observable dependences, induced by a semantic slice:

Definition 8.3.3. Dependences observable in a semantic slice (Example 8.1.1 continued).

Let us consider the observable dependences for the semantic slice, which we defined in Example 8.1.1.
We completely described dependences induced by the criterion $\{(l_5, y)\}$ in Example 8.2.7. The traces in the semantic slice all go through the **true** *branch; as a result, the dependences which were due to the false branch are no longer observable. As a result, we get the following set of observable dependences*

$$\{(l_5, y), (l_2, y), (l_1, x), (l_0, x)\} \times \{(l_5, y)\}$$

Obviously, this set of dependences is significantly smaller than the set of dependences computed in Example 8.2.7.

8.3.4 Approximation of observable dependences

In this section, we still consider a strongly closed semantic slice $\mathcal{E}'$ of a set of traces $\mathcal{E}$.

Approximation of local, observable dependences: As in Section 8.2.3, we define an approximation for local dependences:

Definition 8.3.4. Local, observable dependences.

We let the local observable dependences $\mathfrak{D}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}']$ *be defined by:*

$$\mathfrak{D}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}'] = \bigcup \{\mathfrak{D}_{\mathrm{s}\mathbb{P}\langle l_0 \cdot l_1 \rangle}[\mathcal{E} \mid \mathcal{E}'] \mid l_0, l_1 \in \mathbb{L}\}$$

As usual, only an over-approximation $\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}']$ of $\mathfrak{D}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}']$ can be computed in practice. Since $\mathfrak{D}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}'] \subseteq \mathfrak{D}_{\mathrm{loc}}$ (same proof as Theorem 8.3.3), we can use $\mathfrak{D}_{\mathrm{loc}}$ as an approximation. We show in Section 8.3.5 how to refine $\mathfrak{D}_{\mathrm{loc}}$ into a more precise, yet still safe, over-approximation of $\mathfrak{D}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}']$.

Computable approximations of observable dependences: The fixpoint approximation still holds in the case of the observable dependences:

Theorem 8.3.4. Approximation of observable dependences.

As in Theorem 8.2.5, we let $\Delta_{\mathfrak{D}} = \{((l, x), (l, x)) \mid l \in \mathbb{L}, x \in \mathbb{X}\}$ *and*

$$\begin{array}{rccl} F_{\overline{\mathfrak{D}}} : & \mathfrak{Dep}_{\mathrm{t}} & \rightarrow & \mathfrak{Dep}_{\mathrm{t}} \\ & D & \mapsto & D \cup \mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}'] \circledast D \end{array}$$

(note that the definition of $F_{\overline{\mathfrak{D}}}$ is based on $\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}']$ instead of $\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}$).

Then:

$$\mathfrak{D}_{\mathrm{st}}[\mathcal{E} \mid \mathcal{E}'] = \mathbf{lfp}_{\Delta_{\mathfrak{D}}} F_{\overleftarrow{\mathfrak{D}}}$$

Proof.

Follows the same steps as the proof of Theorem 8.2.5. □

In particular, the computation of the observable dependences induced by a criterion also generalizes straightforwardly (Theorem 8.2.10).

8.3.5 Refining observable dependences

In this section, we consider how to cut down an approximation of the observable dependences induced by the semantic slice $\mathcal{E}'$. Most of the refinements can be applied to when computing the approximation $\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}']$ for the local dependences.

Removal of unreachable control states: In case some control state is unreachable in the semantic slice $\mathcal{E}'$, then it does not appear in any observable dependence of this slice:

Theorem 8.3.5. Dependences and unreachable states.

Let $l, l' \in \mathbb{L}$, $x, x' \in \mathbb{X}$. Then, $((l, x), (l', x')) \in \mathfrak{D}_{\mathrm{st}}[\mathcal{E} \mid \mathcal{E}']$ implies that l and l' are reachable (i.e., there exists a trace $\langle \dots, (l, \rho), \dots \rangle$ in $\mathcal{E}'$, and the same for l').

Proof.

Let us assume that l is not reachable. If we use the same notations as in Definition 8.3.3, then $\mathcal{M}_i = \emptyset$; moreover, we get the result $\mathfrak{D}_{\mathrm{sf}}[\alpha_{\iota\mathcal{F}[l,l']}(\mathcal{E}); \emptyset \mapsto \mathcal{M}_o] = \emptyset$ from the definition of the observable dependences of a function (Definition 8.3.2), since we cannot find two distinct values v_a, v_b in $\mathcal{M}_i(x)$. We conclude that $((l, x), (l', x')) \notin \mathfrak{D}_{\mathrm{st}}[\mathcal{E} \mid \mathcal{E}']$. Similarly, if l' is not reachable, then $\mathcal{M}_o = \emptyset$ and $\mathfrak{D}_{\mathrm{sf}}[\alpha_{\iota\mathcal{F}[l,l']}(\mathcal{E}); \mathcal{M}_i \mapsto \emptyset] = \emptyset$. As a conclusion $((l, x), (l', x')) \notin \mathfrak{D}_{\mathrm{st}}[\mathcal{E} \mid \mathcal{E}']$. □

As a consequence, any non-reachable state does not appear in $\mathfrak{D}_{\mathrm{st}}[\mathcal{E} \mid \mathcal{E}']$ and should not be considered in $\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}']$. If $\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}']$ is a sound over-approximation of $\mathfrak{D}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}']$, then so is the following:

$$\mathfrak{D}^{\mathrm{a}}_{\mathrm{loc}}[\mathcal{E} \mid \mathcal{E}'] \setminus \{((l, x), (l', x')) \in (\mathbb{L} \times \mathbb{X})^2 \mid l \text{ or } l' \text{ is not reachable}\}$$

In practice, such an approximation should be computed by a sound static analysis of the program (prior to dependence analysis).

Removal of constant variables: A similar argument holds for constant variables:

Theorem 8.3.6. Dependences and constant variables.

Let $l \in \mathbb{L}$ and $x \in \mathbb{X}$. If x may take at most one value v at point l in any execution in the semantic slice $\mathcal{E}$, then, there is no dependence to (l, x).

Proof.

Similar as the proof of Theorem 8.3.5: if x is constant at point ℓ, then we cannot find *two distinct values* for x at ℓ and we cannot exhibit a dependence $((\ell, x), (\ell', x'))$. □

Definition 8.3.4. Removal of constant variables (Example 8.2.3 continued.

In the case of the program in Example 8.2.3, $x = 4$ at ℓ_5, for any *execution of the program; as a consequence, the dependence $(\ell_5, x) \rightsquigarrow (\ell_0, b)$ does not hold.*

Constant expressions: If an expression is constant in a semantic slice, then it does not induce any dependence. Indeed, if e always evaluates to the same value v, then its value depends on nothing, so we can provide a better approximation for the dependences induced by an assignment or a condition than the result of Lemma 8.2.7. Of course, this refinement also applies to sub-expressions. Note that this refinement somewhat extends the previous one (removal of constant variables).

For instance, if static analysis proves that in the semantic slice under consideration $x = y$, then the assignment $t = u + 2 \star (x - y)$ a dependence $t \rightsquigarrow u$.

Partitioning and dependence analysis: The analysis carried out in the semantic slicing may resort to some kind of trace partitioning: either control-based [MR05], as in Chapter 5 or in order to distinguish execution patterns [Riv05b], as in Section 7.2.3. Then, the same principle could be applied to the dependence analysis. In particular, this approach allows to benefit from precise abstract invariants, so it may increase the number of contexts the above refinements can be applied in.

Definition 8.3.5. Partitioning dependence analysis.

Let us consider the program below:

```
ℓ_0 :  if(b) {x_0 = y}
       else {x_1 = y};
       if(b') {z = x_0}
       else {z = x_1};
ℓ_1 :  ...
```

We focus on the semantic slice collecting all executions going through the same *branch in both* **if** *statements. Then, the partitioning dependence analysis infers only one dependence from (l_1, z), namely (l_0, y).*
The non-partitioning analysis would also include dependences on $(l_0, b), (l_0, b'), (l_0, x_1), (l_0, x_0)$.
We can see that this refinement allows for global precision improvements.

8.4 Abstract Dependences

We propose a further strengthening of the notion of dependences, after introducing the *observable dependences* in Section 8.3. More precisely, we wish to distinguish dependences, which can be observed even if we perform an abstraction of sets of control states: these dependences are *abstract dependences*.

8.4.1 Definition of abstract dependences

Abstractions: When investigating the causes for an alarm, we are not interested in all computations. Only the computations, which may cause an error are relevant.

For instance, if we focus on an alarm corresponding to a possible overflow, we are usually interested in finding out where large values stem from, and how they may propagate in the program. Similarly, if a specification provides normal ranges for the program variables (for instance, the type system of the ADA programming language allows for such information to be mentioned in programs), we may want to search how abnormal values propagate.

As a consequence, we introduce dependences between *abstractions*. In the following, we consider abstractions of sets of values: we write $\mathbb{Abs}$ for the set of such abstractions, which define a Galois-connection [CC77] (Definition 2.3.1). An element of $\mathbb{Abs}$ is a tuple (D, α, γ), defining a Galois connection $(\mathcal{P}(\mathbb{V}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (D, \sqsubseteq)$ (we do not explicit the order when writing an element of $\mathbb{Abs}$, for the sake of concision). For short, we may let $\mathbb{a}_0$ denote the tuple $(D_0, \alpha_0, \gamma_0)$.

Abstract dependences induced by functions: We now embed abstractions into dependences (the case of observable dependences is postponed):

Definition 8.4.1. Abstract dependences.

Let $\mathbb{a}_0 = (D_0, \alpha_0, \gamma_0)$ *and* $\mathbb{a}_1 = (D_1, \alpha_1, \gamma_1)$ *be two abstractions,* $x_0, x_1 \in \mathbb{X}$*, and* $\phi \in \mathfrak{Den}$*. We say that there is an* abstract dependence *induced by* ϕ *of* $(x_1, \mathbb{a}_1)$ *on* $(x_0, \mathbb{a}_0)$ *if and only if:*

$$\exists \rho \in \mathbb{M},\ \exists d_a, d_b \in D_0,\ \left\{ \begin{array}{ll} \multicolumn{2}{l}{\alpha_1(\phi(\rho_a)(x_1)) \neq \alpha_1(\phi(\rho_b)(x_1))} \\ \textit{where} & \gamma_0(d_i) \neq \emptyset \\ \textit{and} & \rho_i = \rho[x_0 \leftarrow \gamma_0(d_i)] \end{array} \right.$$

Such a dependence will be denoted by $(x_1, \mathbb{a}_1) \overset{\phi}{\rightsquigarrow} (x_0, \mathbb{a}_0)$.
Furthermore, we let the abstract dependence set $\mathfrak{D}^\sharp_{\mathrm{f}}[\phi]$ *of* ϕ *be defined by:*

$$\mathfrak{D}^\sharp_{\mathrm{f}}[\phi] = \{((x_0, \mathbb{a}_0), (x_1, \mathbb{a}_1)) \in (\mathbb{X} \times \mathbb{Abs})^2 \mid (x_1, \mathbb{a}_1) \overset{\phi}{\rightsquigarrow} (x_0, \mathbb{a}_0)\}$$

Intuitively, $(x_1, \mathbb{a}_1)$ depends on $(x_0, \mathbb{a}_0)$ if substituting to x_0 two values which can be distinguished by α_0 results in x_1 having different values, distinguished by α_1 after the execution of ϕ. The following example illustrate the usefulness of the approach:

Definition 8.4.1. Abstract dependences of a function.

In this example, we consider that the set of values is the set of natural integers $\mathbb{Z}$ and that the abstraction $\mathbb{a} = (D, \alpha, \gamma)$ is defined by $D = \{\bot, d_0, d_1, \top\}$, and:

$$\gamma : \begin{cases} \bot & \mapsto & \emptyset \\ d_0 & \mapsto & \{v \in \mathbb{Z} \mid |x| < 1\,000\} \\ d_1 & \mapsto & \{v \in \mathbb{Z} \mid |x| \geq 1\,000\} \\ \top & \mapsto & \mathbb{Z} \end{cases}$$

Let us focus on the function:

$$\begin{array}{rccl} \phi : & \mathbb{M} & \longrightarrow & \mathbb{M} \\ & \rho & \mapsto & \rho[z \leftarrow (x \mod 2) \star y] \end{array}$$

In the standard settings of Definition 8.2.1, ϕ induces two dependences $z \rightsquigarrow x$ and $z \rightsquigarrow y$. However, if we consider abstract dependences, though the situation is rather different:

- *if we let $\rho(x) = 1$, then $\phi(\rho)(z) = \rho(y)$, so that we can verify straightforwardly that ϕ induces an abstract dependence $(z, \mathbb{a}) \rightsquigarrow (y, \mathbb{a})$;*
- *however, whether the value of x is large or not does not affect the output of ϕ so that $(z, \mathbb{a})$ does not depend on $(x, \mathbb{a})$.*

Of course, we may consider different abstractions instead of $\mathbb{a}$, and get different results. For instance, if $\mathbb{a}'$ is the parity abstraction, then $(z, \mathbb{a}) \overset{\phi}{\rightsquigarrow} (x, \mathbb{a}')$.

We now prove that this definition of abstract dependences generalizes "concrete" dependences, which we introduced in Definition 8.2.1.

Theorem 8.4.1. Dependences are abstract dependences.

We write id *for the identity abstraction, i.e. the tuple (D, α, γ) characterized by $D = \mathcal{P}(\mathbb{V})$, $\alpha = \gamma = \lambda(X \in \mathcal{P}(\mathbb{V})) \cdot X$.*
Let $\phi \in \mathfrak{Den}$, $x_0, x_1 \in \mathbb{X}$. Then:

$$x_1 \overset{\phi}{\rightsquigarrow} x_0 \iff (x_1, \mathrm{id}) \overset{\phi}{\rightsquigarrow} (x_0, \mathrm{id})$$

Proof.

Let us assume that $x_1 \overset{\phi}{\rightsquigarrow} x_0$. Then, there exist $\rho \in \mathbb{M}$, $v_a, v_b \in \mathbb{V}$, such that $\phi(\rho_a)(x_1) \neq \phi(\rho_b)(x_1)$, where $\forall i,\ \rho_i = \rho[x_0 \leftarrow v_i]$. Let $d_i = \{v_i\}$. Then, clearly, $\forall i,\ \{\rho_i\} = \rho[x_0 \leftarrow \mathrm{id}(d_i)]$ and $\mathrm{id}(\phi(\rho_a)(x_1)) \neq \mathrm{id}(\phi(\rho_b)(x_1))$, which proves that $(x_1, \mathrm{id}) \overset{\phi}{\rightsquigarrow} (x_0, \mathrm{id})$.

Let us assume that $(x_1, \mathrm{id}) \overset{\phi}{\rightsquigarrow} (x_0, \mathrm{id})$. Then, there exist $\rho \in \mathbb{M}$, $d_a, d_b \in \mathcal{P}(\mathbb{V})$, such that $\mathrm{id}(\phi(\rho_a)(x_1)) \neq \mathrm{id}(\phi(\rho_b)(x_1))$ where $\forall i,\ \rho_i = \rho[x_0 \leftarrow d_i]$, and $\forall i,\ d_i \neq \emptyset$. Then, $\exists v \in \phi(\rho_a)(x_1),\ v \notin \phi(\rho_b)(x_1)$ (or the converse holds and we may just permute a and b and recover the above statement). As a consequence, $\exists v_a \in \mathbb{V},\ v \in \phi(\rho[x_0 \leftarrow v_a])(x_1)$ (since $\phi(\rho_a) = \{\phi(\rho[x_0 \leftarrow v]) \mid v \in d_a\}$). We can pick up any $v_b \in d_b$. Clearly, $v \notin \phi(\rho[x_0 \leftarrow v_b])(x_1)$. This shows that $x_1 \overset{\phi}{\rightsquigarrow} x_0$. □

Abstract observable dependences: We generalize the notion of observable dependences in the same way:

Definition 8.4.2. Abstract dependences.

Let $\phi \in \mathfrak{Den}$, $\mathcal{M}_i, \mathcal{M}_o \subseteq \mathbb{M}$, $x_0, x_1 \in \mathbb{X}$, $\alpha_0, \alpha_1 \in \mathbb{Abs}$. We say that ϕ induces an abstract dependence *of $(x_1, \mathfrak{a}_1)$ on $(x_0, \mathfrak{a}_0)$ in the semantic slice defined by $(\mathcal{M}_i, \mathcal{M}_o)$ if and only if:*

$$\exists \rho \in \mathcal{M}_i,\ d_a, d_b \in D_0, \left\{ \begin{array}{l} \alpha_1(\phi(\rho_a)(x_1) \cap \mathcal{M}_o(x_1)) \neq \alpha_1(\phi(\rho[x_0 \leftarrow v_b])(x_1) \cap \mathcal{M}_o(x_1)) \\ \textit{where} \quad \gamma_0(d_i) \cap \mathcal{M}_i(x_0) \neq \emptyset \\ \textit{and} \qquad \rho_i = \rho[x_0 \leftarrow \gamma_0(d_i) \cap \mathcal{M}_i(x_0)] \end{array} \right.$$

We write $(x_1, \mathfrak{a}_1) \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \mapsto \mathcal{M}_o} (x_0, \mathfrak{a}_0)$ if such a dependence holds.
Furthermore, we let the abstract dependence set $\mathfrak{D}^\sharp_{\mathrm{sf}}[\phi; \mathcal{M}_i \mapsto \mathcal{M}_o]$ *of ϕ be defined by:*

$$\mathfrak{D}^\sharp_{\mathrm{sf}}[\phi; \mathcal{M}_i \mapsto \mathcal{M}_o] = \{((x_0, \mathfrak{a}_0), (x_1, \mathfrak{a}_1)) \in (\mathbb{X} \times \mathbb{Abs})^2 \mid (x_1, \mathfrak{a}_1) \overset{\phi}{\rightsquigarrow}_{\mathcal{M}_i \mapsto \mathcal{M}_o} (x_0, \mathfrak{a}_0)\}$$

As we can see, this definition is obtained directly as a generalization of the definitions for observable (Definition 8.3.2) and abstract (Definition 8.4.1) dependences. In the following, we consider abstract dependences only, so as to make the presentation more simple; however, all the results presented here generalize to the case of observable, abstract dependences.

We can note that abstract dependences are a kind of dual of the notion of abstract non-interference [GM04], even though [GM04] provides several definitions of "abstract" secrecy and none of them clearly corresponds to our settings: it seems closer to the notion of $\langle \eta, \rho, \phi \rangle$-Secrecy, where η, ρ and ϕ respectively denote the abstraction applied to the low inputs (in our case, the identity), the abstraction applied to the low outputs (in our case, the observation of the results is defined by both the observation $\mathcal{M}_i$ and the abstraction $\mathfrak{a}_1$) and ϕ is the abstraction on the high inputs (in our case $\mathfrak{a}_0$). Though, the motivation for abstract non-interference is rather different than ours, and most of the methods presented in [GM04] aim at proving secrecy or discovering for what domains secrecy holds. By contrast, we focus on computing relevant sets of dependences (and not proving the absence of dependences).

Definition for sets of traces: We derive the abstract dependences of a set of traces from the abstract dependences induced by a function as usual, i.e., by applying the definition of function dependences to denotational abstractions of the set of traces.

As a consequence, we focus on a strongly closed set of traces $\mathcal{E}$:

Definition 8.4.3. Abstract dependences –case of sets of traces.

Let $l_0, l_1 \in \mathbb{L}$, $\mathbb{a}_0, \mathbb{a}_1 \in \mathbb{Abs}$ and $x_0, x_1 \in \mathbb{X}$. Then, we say that $\mathcal{E}$ induces an abstract dependence *of $(l_0, x_0, \mathbb{a}_0)$ on $(l_1, x_1, \mathbb{a}_1)$ if and only if:*

$$((x_0, \mathbb{a}_0), (x_1, \mathbb{a}_1)) \in \mathfrak{D}^\sharp_{\mathrm{f}}[\alpha_{\iota\mathcal{F}[l_0,l_1]}(\mathcal{E})]$$

As usual, we let $(l_1, x_1, \mathbb{a}_1) \overset{\phi}{\rightsquigarrow} (l_0, x_0, \mathbb{a}_0)$ denote such a dependence.
Moreover, we write $\mathfrak{D}^\sharp_{\mathrm{t}}[\mathcal{E}]$ for the abstract dependence set *defined by:*

$$\mathfrak{D}^\sharp_{\mathrm{t}}[\mathcal{E}] = \{((l_0, x_0, \mathbb{a}_0), (l_1, x_1, \mathbb{a}_1)) \mid (l_1, x_1, \mathbb{a}_1) \overset{\phi}{\rightsquigarrow} (l_0, x_0, \mathbb{a}_0)\}$$

We can now restrict even further the dependences induced by the semantic slice of Example 8.1.1:

Definition 8.4.2. Example 8.2.7 revisited.

Let us inspect again the dependences of the program displayed in Figure 8.1(a). All dependences from (l_5, y) were listed in Example 8.2.7, and we expect to cut down these dependences a little bit, by restricting to abstract dependences, corresponding to the abstract $\mathbb{a}$ which we introduced in Example 8.4.1.
In fact, the only abstract observable dependence from $(l_5, y, \mathbb{a})$ is $(l_5, y, \mathbb{a}) \rightsquigarrow (l_2, y, \mathbb{a})$. Indeed, there is no dependence in the **false** *branch (since it is unreachable in the semantic slice); moreover, the x may not take any large value.*

8.4.2 Hierarchies of dependences

In the same way as we could compare sets of observable dependences corresponding to comparable observations, we can also state a similar "monotonicity" result in the case of abstract dependences. Intuitively, the existence of an abstract dependence implies the existence of a dependence for any pair of more concrete abstractions. In other words, abstract dependences express a stronger property than mere dependences.

We prove this result in the settings of Definition 8.4.1, i.e., we do not consider observable abstract dependences here. We would deal with the observable abstract dependences in a similar way.

Theorem 8.4.2. Abstract dependences hierarchy.

Let $\mathbb{a}_0 = (D_0, \alpha_0, \gamma_0)$, $\mathbb{a}'_0 = (D'_0, \alpha'_0, \gamma'_0)$, $\mathbb{a}_1 = (D_1, \alpha_1, \gamma_1)$, $\mathbb{a}'_1 = (D'_1, \alpha'_1, \gamma'_1)$ *be four abstractions, such that there exist two Galois connections:*

$$D'_0 \underset{\alpha''_0}{\overset{\gamma''_0}{\leftrightarrows}} D_0 \qquad D'_1 \underset{\alpha''_1}{\overset{\gamma''_1}{\leftrightarrows}} D_1$$

and such that

$$\begin{array}{cc} \alpha_0 = \alpha''_0 \circ \alpha'_0 & \alpha_1 = \alpha''_1 \circ \alpha'_1 \\ \gamma_0 = \gamma'_0 \circ \gamma''_0 & \gamma_1 = \gamma'_1 \circ \gamma''_1 \end{array}$$

We assume that $(x_1, \mathbb{a}_1) \overset{\phi}{\rightsquigarrow} (x_0, \mathbb{a}_0)$. *Then, the following dependences hold:*

1. $(x_1, \mathbb{a}'_1) \overset{\phi}{\rightsquigarrow} (x_0, \mathbb{a}_0)$;
2. $(x_1, \mathbb{a}_1) \overset{\phi}{\rightsquigarrow} (x_0, \mathbb{a}'_0)$;
3. $(x_1, \mathbb{a}'_1) \overset{\phi}{\rightsquigarrow} (x_0, \mathbb{a}'_0)$.

Proof.

By assumption, $(x_1, \mathbb{a}_1) \overset{\phi}{\rightsquigarrow} (x_0, \mathbb{a}_0)$, hence, there exist $\rho \in \mathbb{M}, d_a, d_b \in D_0$, such that $\alpha_1(\phi(\rho_a)(x_1)) \neq \alpha_1(\phi(\rho_b)(x_1))$ where $\rho_i = \rho[x_0 \leftarrow \gamma_0(d_i)]$ and $\gamma_0(d_i) \neq \emptyset$. We prove the first two points under that assumption:

1. if $\alpha'_1(\phi(\rho_a)(x_1)) = \alpha'_1(\phi(\rho_b)(x_1))$, then $\alpha_1(\phi(\rho_a)(x_1)) = \alpha''_1 \circ \alpha'_1(\phi(\rho_a)(x_1)) = \alpha''_1 \circ \alpha'_1(\phi(\rho_b)(x_1)) = \alpha_1(\phi(\rho_b)(x_1))$, which does not hold; so $\alpha'_1(\phi(\rho_a)(x_1)) \neq \alpha'_1(\phi(\rho_b)(x_1))$, which proves the first point.
2. we let $d'_i = \gamma''_0(d_i)$; then $\rho_i = \rho[x_0 \leftarrow \gamma'_0(d'_i)]$, since $\gamma_0 = \gamma'_0 \circ \gamma''_0$; hence, there exist $d'_a, d'_b \in D'_0$ that satisfy the definition of abstract dependence; since $\gamma'_0(d'_i) \neq \emptyset$ (otherwise, we would have $\gamma_0(d_i) = \gamma'_0(d'_i) = \emptyset$), this proves the second point.

The third point follows from the above two points; it can be proved in two steps (applying abstraction on the left side, then on the right side of the dependence arrow). □

At this point, we have a full hierarchy of dependences:

- mere dependences correspond to the negation of non-interference;
- observable dependences are dependences which can be observed, even if only a subset of the traces is available; furthermore, the smaller the semantic slice, the fewer dependences we can observe on it (if the slice contains all the traces, then all dependences are observable);
- abstract dependences are dependences which can be observed, even if we can distinguish an abstraction of values (and not just values); moreover, the more abstract the abstractions, the fewer dependences we can observe through it.

Definition 8.4.3. Hierarchy of dependences.

We sum up the various kinds of dependences induced by the program displayed in Figure 8.1(a) (Example 8.1.1). In particular, only one abstract observable dependence remains, which points directly to the line where y is assigned a large value, and also where a large

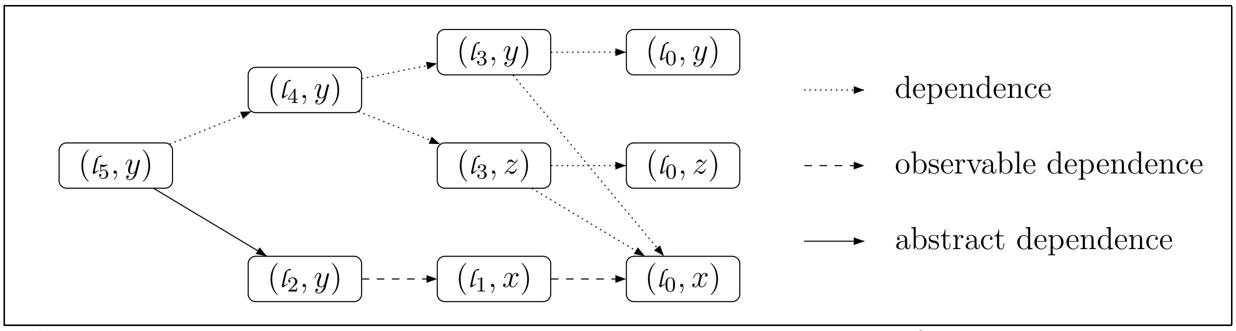


Figure 8.3: Local dependences involved in the approximation of the backward dependence induced by $\{(\ell_5, y)\}$

value appears for the first time in the execution of the program. Consequently, this notion of dependence turns out to be adequate in order to find out the origin of the large value for y at point ℓ_5.

8.4.3 Approximation of abstract dependences

We showed the notion of abstract dependence to be useful for the investigation of alarms; however, we need to set up algorithms for this notion to be really of any practical interest. Therefore, we propose to extend the fixpoint algorithms.

Approximation of composition: We let $\mathfrak{Dep}^{\sharp}_{\mathfrak{f}}$ denote $\mathcal{P}((\mathbb{X} \times \mathbb{Abs})^2)$. We define the dependence composition operator as usual:

Definition 8.4.4. Composition of abstract dependences.
Let $\mathfrak{D}_0, \mathfrak{D}_1 \in \mathfrak{Dep}^{\sharp}_{\mathfrak{f}}$. Then, we let $\mathfrak{D}_0 \boxtimes \mathfrak{D}_1$ be defined by

$$\begin{array}{rcl}\mathfrak{D}_0 \boxtimes \mathfrak{D}_1 & = & \{((x_0, \mathbb{a}_0), (x_2, \mathbb{a}_2)) \in (\mathbb{X} \times \mathbb{Abs})^2 \mid \exists (x_1, \mathbb{a}_1) \in \mathbb{X} \times \mathbb{Abs} \\ & & \quad ((x_0, \mathbb{a}_0), (x_1, \mathbb{a}_1)) \in \mathfrak{D}_0 \wedge ((x_1, \mathbb{a}_1), (x_2, \mathbb{a}_2)) \in \mathfrak{D}_1\}\end{array}$$

We note that this composition operator can be defined more simply, when applied to dependences of functions:

Theorem 8.4.3. Alternate definition of $\boxtimes$.
Let $\phi_0, \phi_1 \in \mathfrak{Dep}^{\sharp}_{\mathfrak{f}}$. Then:

$$\begin{array}{rcl}\mathfrak{D}^{\sharp}_{\mathfrak{f}}[\phi_0] \boxtimes \mathfrak{D}^{\sharp}_{\mathfrak{f}}[\phi_1] & = & \{((x_0, \mathbb{a}_0), (x_2, \mathbb{a}_2)) \in (\mathbb{X} \times \mathbb{Abs})^2 \mid \exists x_1 \in \mathbb{X} \\ & & \quad ((x_0, \mathbb{a}_0), (x_1, \mathrm{id})) \in \mathfrak{D}^{a}_{f}[\phi_0] \wedge ((x_1, \mathrm{id}), (x_2, \mathbb{a}_2)) \in \mathfrak{D}^{a}_{f}[\phi_1]\}\end{array}$$

Proof.

It is easy to prove a double inclusion:

- the definition in Theorem 8.4.3 is clearly included in the one introduced in Definition 8.4.4;
- the converse inclusion follows from the result of Theorem 8.4.2: if $(x_1, \mathbb{a}_1) \overset{\phi_0}{\rightsquigarrow} (x_0, \mathbb{a}_0)$, then $(x_1, \mathrm{id}) \overset{\phi_0}{\rightsquigarrow} (x_0, \mathbb{a}_0)$ (a similar result holds for ϕ_1).

The theorem follows. □

The soundness of $\boxtimes$ with respect to $\circ$ follows:

Theorem 8.4.4. Composition of abstract dependences –approximation.

Let $\phi_0, \phi_1 \in \mathfrak{Den}$. *Then:*

$$\mathfrak{D}^\sharp_{\mathfrak{f}}[\phi_1 \circ \phi_0] \subseteq \mathfrak{D}^\sharp_{\mathfrak{f}}[\phi_0] \boxtimes \mathfrak{D}^\sharp_{\mathfrak{f}}[\phi_1]$$

Proof.

Using the alternate definition for $\boxtimes$ when applied to dependence sets, the proof of the theorem follows the same steps as the proof of Theorem 8.2.2. □

In fact, the conclusion of Theorem 8.4.3 (and the fact that it plays a great role in the proof of Theorem 8.4.4) hides a major weakness in this approximation of the function composition. Indeed, it means that the approximate abstract dependences computed when considering a path p will also include mere, concrete dependences.

Fixpoint-based approximation: Even though we pointed out a significant issue with the approximation of $\circ$, we state the fixpoint-based approximation for abstract dependences (a deeper study will reveal other drawbacks, and allow for an alternate method to be stated).

We assume that $\mathfrak{D}^\sharp_{\mathrm{loc}}$ over-approximate the abstract local dependences and that $\boxbar$ extends $\boxtimes$ to $\mathfrak{Dep}^\sharp_{\mathfrak{t}} = \mathcal{P}((\mathbb{L} \times \mathbb{X} \times \mathbb{Abs})^2)$. Furthermore, we let $\Delta^\sharp_{\mathfrak{D}} = \{((\ell, x, \mathbb{a}), (\ell, x, \mathbb{a})) \mid (\ell, x, \mathbb{a}) \in \mathbb{L} \times \mathbb{X} \times \mathbb{Abs}\} \in \mathfrak{Dep}^\sharp_{\mathfrak{t}}$, and:

$$\begin{array}{rccl} F^\sharp_{\overleftarrow{\mathfrak{D}}} : & \mathfrak{Dep}^\sharp_{\mathfrak{t}} & \rightarrow & \mathfrak{Dep}^\sharp_{\mathfrak{t}} \\ & D & \mapsto & D \cup \mathfrak{D}^\sharp_{\mathrm{loc}} \boxbar D \end{array}$$

Theorem 8.4.5. Fixpoint approximation of abstract dependences.

$$\mathfrak{D}^\sharp_{\mathfrak{t}}[\mathcal{E}] \subseteq \mathbf{lfp}_{\Delta^\sharp_{\mathfrak{D}}} F^\sharp_{\overleftarrow{\mathfrak{D}}}$$

Proof.
Similar as the proof of Theorem 8.2.5. □

However, this theorem does not give an effective way of computing a precise approximation of the set of abstract dependences since bounding precisely the local dependences presents several great difficulties:

- $\mathbb{A}\mathrm{bs}$ is *not countable* and *not computer representable*, or has a prohibitive size even if the number of possible values is finite and small. As a consequence, some kind of approximation is necessary (it could be justified by the result on the hierarchy of abstract dependences).
- the *hierarchy result does not apply straightforwardly*: proving that there is no dependence $(x_0, \mathfrak{a}_0) \overset{\phi}{\rightsquigarrow} (x_1, \mathfrak{a}_1)$ tells nothing about a dependence $(x_0, \mathfrak{a}_0') \overset{\phi}{\rightsquigarrow} (x_1, \mathfrak{a}_1')$, where for instance $\mathfrak{a}_0'$ is *more concrete* than $\mathfrak{a}_0$, but $\mathfrak{a}_1'$ is *more abstract* than (or not comparable to) $\mathfrak{a}_1$.
- the dramatic precision issue encountered in the approximation for $\circ$ also prevents from computing relevant abstract dependences (i.e., from refining the classical dependences).

Overall, these issues stem from the nature of the problem, which the least-fixpoint result of Theorem 8.4.5 tackles. Indeed, $\mathfrak{D}_{\mathrm{t}}^{\sharp}[\mathcal{E}] \cap (\mathbb{L} \times \mathbb{X} \times \mathbb{A}\mathrm{bs}) \times \{(l_0, x_0, \mathfrak{a}_0)\}$ collects all the tuples which may affect the observation of the abstraction $\mathfrak{a}_0$ of x_0 at point l_0; in particular it includes *all kinds of properties*, which may affect this observation. This is far beyond what we wish to achieve in priority: our purpose is to find out the *immediate* causes for some event (such as an error) to occur. As a consequence, we propose to narrow our setup.

8.4.4 Chains of abstract dependences

Restriction to dependence chains: We adopt the following restrictions, so as to compute relevant abstract dependences:

- restrict to *some set* of abstractions $\mathfrak{a}$: not all abstractions are intuitive or informative (for instance, we may focus on abstraction discriminating "large values");
- limit the closure to a chain of *immediate* causes, which may affect the criterion (so that, more intricate causes should not be considered, at least in a first approach).

These restriction lead us to the notion of *dependence chains*:

Definition 8.4.5. Abstract dependence chain.

A dependence chain *is a sequence* $(l_0, x_0, \mathfrak{a}_0), \ldots, (l_n, x_n, \mathfrak{a}_n)$ *of elements of* $\mathbb{L} \times \mathbb{X} \times \mathbb{A}\mathrm{bs}$*, such that:*

$$\forall i,\ ((l_i, x_i, \mathfrak{a}_i), (l_{i+1}, x_{i+1}, \mathfrak{a}_{i+1})) \in \mathfrak{D}_{\mathrm{loc}}^{\sharp}$$

Let $\mathfrak{a} \subseteq \mathbb{A}\mathrm{bs}$*. Then, we say that the chain* $(l_0, x_0, \mathfrak{a}_0), \ldots, (l_n, x_n, \mathfrak{a}_n)$ *is* $\mathfrak{a}$-abstract *if* $\forall i,\ \mathfrak{a}_i \in \mathfrak{a}$*.*

Obviously, in case $(l_0, x_0, \mathfrak{a}_0), \ldots, (l_n, x_n, \mathfrak{a}_n)$ is an $\mathfrak{a}$-abstract dependence chain for $\mathcal{E}$, there exists a dependence $(l_n, x_n, \mathfrak{a}_n) \overset{\mathcal{E}}{\rightsquigarrow} (l_0, x_0, \mathfrak{a}_0)$. However, the converse does not hold true. It may be the case that no $\mathfrak{a}$-abstract chain exists between $(l_0, x_0, \mathfrak{a}_0)$ and $(l_n, x_n, \mathfrak{a}_n)$, but there exists a non $\mathfrak{a}$-abstract chain on the same path. Therefore, the computation of abstract dependence chains is not a solution for *over*-approximating dependences; it is at most useful for providing an under-approximation (defined by the two restrictions mentioned in the beginning of this subsection).

Computation of dependence chains: The computation of all the $\mathfrak{a}$-abstract dependence chains from a criterion $(l_0, x_0, \mathfrak{a}_0) \in \mathbb{L} \times \mathbb{X} \times \mathbb{A}\mathrm{bs}$ can be achieved via a least-fixpoint algorithm similar to the one proposed in Theorem 8.2.5. However, we should use an approximation of the local $\mathfrak{a}$-abstract dependences. Dependences provide such an approximation. We propose to improve this rough approximation with refinements, as we did in Section 8.3.5.

Refinements: We assume that we consider the $\mathfrak{a}$-abstract dependences of a semantic slice $\mathcal{E}'$ of $\mathcal{E}$ (Definition 8.4.2).

Then, all the refinements introduced in Section 8.3.5 apply, since abstract dependences are a subset of dependences (hence, if we can prove that there is no dependence between (l_0, x_0) and (l_1, x_1), then there is no abstract dependence either).

Moreover, we can also propose an abstract version of the "removal of constant variables" in the case of abstract dependences. We assume that we have computed an approximation $\mathcal{E}'^{\sharp} \in \mathbb{L} \to (\mathbb{X} \to \mathcal{P}(\mathbb{V}))$ of the semantic slice $\mathcal{E}'$ (Chapter 7). Let us consider $(l_0, x_0, \mathfrak{a}_0), (l_1, x_1, \mathfrak{a}_1) \in \mathbb{L} \times \mathbb{X} \times \mathbb{A}\mathrm{bs}$. If there exists a minimal element d_0 of $D_0 \backslash \{\bot\}$ (where $\bot$ is the least element of D_0) such that $\mathcal{E}'^{\sharp}(l_0)(x_0) \subseteq \gamma_0(d_0)$, then the abstract domain D_0 is not able to distinguish the values observed for x_0 at l_0 in the semantic slice. An obvious application of Definition 8.4.1 shows that there is no dependence $(l_1, x_1, \mathfrak{a}_1) \leadsto_{[\mathcal{E}']} (l_0, x_0, \mathfrak{a}_0)$. For instance, this refinement applies if $\mathfrak{a}_0$ abstracts together all "normal" (i.e., not too large) values and if all values for x_0 at point l_0 are "normal".

Definition 8.4.4. Abstract dependence chains.

In the case of the program presented in Figure 8.1(a) (Example 8.1.1), the above refinement allows to restrict the set of abstract dependences from (l_5, y) to the only dependence (l_2, y), i.e., to recover the result displayed in Figure 8.3.

As a consequence, there is only one $\mathfrak{a}$-abstract dependence chain from (l_5, y), and it leads to (l_2, y), which turns out to be the point where an "abnormal" value appears for the first time in the sequence of computations leading to y, due to x being multiplied by a large number. In this example, we remark that abstract dependence chains are effective as a means to track a special kind of error.

8.5 Abstract Slices

Slicing: Slicing [Wei81] aims at selecting a subset of the statements of a program that may play a role in the computation of some variable x at some point l. The principle is to include in the slice any statement at point l' that may modify a variable x' such that (l, x) depends on (l', x').

The semantics of program slicing is rather subtle for several reasons:

- The notion of dependence involved in slicing is quite different to the one we considered in Section 8.2. For instance the slice of $l_0 : x = 3; l_1 : y = x; l_2$ for the criterion (l_2, y) should include the statement $l_0 : x = 3; l_1$ as well, even though (l_2, y) does not depend on (l_1, x) according to Definition 8.3.3, since x is constant at l_1.
- The usual expression of slicing correctness resorts to some kind of projection of the program semantics (Section 3.4, which is preserved by slicing. However, the removal of non-terminating loops (or of possible sources for errors) may cause the slice to present *more* behaviors than the projection of the semantics of the source program. This issue can be solved by considering a non-standard lazy semantics [CF89], which is preserved by the transformation, yet this approach is not natural for static analysis.

As a consequence, we propose a transformation that should be more adapted to static analysis, and to the discovery of the origin of alarms.

Smaller, non-executable slices: The semantic slices introduced in Chapter 7 approximate program executions with abstract invariants. Such an invariant together with a (subset of a) syntactic slice allow to describe even more precisely a set of program executions:

Definition 8.5.1. Abstract slice.

An abstract slice $\mathfrak{S}^\sharp$ of a program s is defined by a sound invariant $\mathbb{I}^\sharp_{\mathfrak{S}} : \mathbb{L} \to \mathcal{P}(\mathbb{M})$ for $\mathfrak{S}^\sharp$ and a subset s' of the program statements, which is defined by the set of corresponding control states $\mathbb{L}_{\mathfrak{S}}$.

The semantics of a semantic slice is defined both by the program transitions (for the statements which are included in the slice) and by the abstract invariants:

Definition 8.5.2. Abstract slice semantics.

The semantics $[\![s']\!]^\sharp_{\mathfrak{S}}$ of the abstract slice collects all the traces $\langle (l_0, \rho_0), \ldots, (l_n, \rho_n) \rangle$ such that:

- $\forall i,\ \rho_i \in \mathbb{I}^\sharp_{\mathfrak{S}}(l_i)$;
- $\forall i,\ (l_i \in \mathbb{L}_{\mathfrak{S}} \wedge l_{i+1} \in \mathbb{L}_{\mathfrak{S}} \wedge (l_i, \rho_i) \to (l_{i+1}, \rho_{i+1})) \Longrightarrow (l_i, \rho_i) \to (l_{i+1}, \rho_{i+1})$.

Obviously, the definition of abstract slices leaves the choice of the syntactic slice undetermined. However, the purpose of the abstract slices is to restrict to the most interesting parts the program; hence, we propose to compute abstract dependence chains and include any assignment which affect a variable in a dependence chain: this way, the slice preserves only the $\mathfrak{a}$-abstract dependence chains and abstract any other statement of the program into the invariants in $\mathfrak{S}^\sharp$. Let us note that this notion allows to solve the two points mentioned above:

- parts of the program that are not immediately relevant to the criterion under investigation (in the sense that they do not appear in the dependences introduced in Definition 8.2.1, Definition 8.3.2 and Definition 8.4.1) do *not* need to be included into the slice anymore; instead, they can be replaced with program invariants (in the semantic slice). For instance, the assignment $l_0 : x = 3; l_1$ can be replaced with the invariant $x = 3$ at point l_1. Obviously, applying this principle to larger programs may result in huge gain in slice sizes. Furthermore, the loss in precision might be limited if we use precise, relational invariants.
- the intersection with program invariants limits the loss of precision induced by, e.g. the removal of a loop.

Definition 8.5.1. Abstract slice.

Let us consider the program of Figure 8.1(a), together with its input/output conditions. Figure 8.3 displays the local, observable and abstract dependences that can be recursively composed when starting from (l_5, y). In case we compute an abstract slice for this program, starting from (l_5, y), we find only one $\mathfrak{a}$-abstract dependence chain (Example 8.4.4). As a consequence, we get the abstract slice defined by the set of control states $\mathbb{L}_{\mathfrak{S}} = \{l_1, l_2, l_5\}$. In particular, the abstract slice contains the assignment $l_1 : y = 1000 \star x; l_2$, with the invariant $(x \in [5, 10])$, which gives a likely cause for the error.

8.6 Implementation and conclusion

8.6.1 Case study

We implemented a dependence analysis and procedures to refine them into observable abstract dependences in Astrée (for tracking large values and overflows), together with an abstract slice extraction algorithm.

We chose to modify some 70 kLOC real world application, so as to make some retroactions unstable (Astrée proves the absence of overflow in the original version). The purpose of this early experiment was to check the ability of the abstract dependence analysis to track where overflows were coming from.

The static analysis by Astrée takes roughly 20 minutes and uses 500 Mb on a Biopteron 2.2 Ghz with 8 Gb of RAM. The computation of the dependence graph (by collecting all local dependences and applying local refinements) takes 72 seconds and requires 300 Mb, on the same machine; this phase provides all data required to extract a slice from any criterion. The slice extraction computes a least fixpoint from the criterion (Theorem 8.2.10) and applies recursively local dependences; in the case of abstract dependences, this amounts to collecting $\mathfrak{a}$-abstract dependence chains. The typical slice extraction time is about 5 seconds, with low memory requirements (around 110 Mb).

The table below displays the gain in size obtained by computing abstract slices for a series of alarms (size of slices are in LOCs):

Slicing point	a_1	a_2	a_3
Classical slice	543	368	1572
Abstract slice	39	160	96

The resulting slices proved helpful for finding the direct consequences of errors like overflows; moreover, it seemed promising for deriving automatically semantic slicing criteria, which was one of the motivations for our present work.

We remarked that the refinements presented in Section 8.3.5 played a great role in keeping the size of dependences down.

Cyclic abstract dependence chains suggest some kind of partitioning could be done in order to isolate certain execution patterns; they also allow to restrict the part of the program to look at in order to define an adequate input for defining an error scenario, so that we envisage synthesizing input constraints in the future. Another possible use for abstract slices is to cut down the size of programs to analyze during alarm inspection sessions, by abstracting into invariants parts of the code to analyze.

8.6.2 Comparison with related work

We proposed a framework for defining and computing valuable dependence information, for the understanding and refinement of static analysis results. Early experiments [Riv05a] back-up favorably the usefulness of this settings, so that we can safely expect it to provide good hints for the choice of semantic slicing criteria [Riv05b].

Our definition for dependences are rather related to the definition of non-interference [GM82] commonly used in language-based security [SM03]. This approach is rather different to the more traditional ways of defining dependences in program slicing, which rely on program dependence graphs [HRB90, HRB88], yet these two problems are related [ABHR99, Aba99]. We found that the main benefit of the “dependences as interference” definition is to allow for wide varieties of refinements for dependence analyses and extension for the definition of dependences to be stated, proved correct and implemented.

Moreover, our definition of abstract dependences is closely related to the notion of abstract non interference introduced in [GM04] in the security area, which aims at classifying program attackers as abstract-interpretations. The authors of [GM04] propose to compute the strongest safe attacker of a program by resolving an equation on domains by fixpoint. In our settings, the abstraction on the output is fixed by the kind of alarm being investigated; moreover, the dependence analysis should discover the variables the criterion depends on and not only for what observation. As a result, we noticed that the algorithms proposed in [GM04] do not apply to our goal, even though the notion is closely related. Development in both areas should be related in the future.

Program slicing [Wei81] is another area related to our work. Many alternative notions of slices [HDSS96] have been proposed since the very first, syntactic versions of slicing. In particular, conditioned slicing [CCL98] aim at extracting slices preserving *some* executions of programs, specified by, e.g., a relation on inputs. Our approach goes beyond these methods: indeed, a set of program executions defined by a semantic property (e.g., leading to an error) is characterized precisely by semantic slicing [Riv05b]; these invariants allow to refine precisely the dependences. Dynamic slicing [KL88, FDHH04] records states during *concrete* executions and inserts a dependence among the corresponding nodes according to a standard, rough dependence analysis, in order to produce "dynamic", non-executable slices. This approach is adapted to debugging; yet it does not allow to characterize precisely a set of executions defined by semantic constraints either.

There exist a wide variety of methods applied to error cause localization. For instance, [BNR03] proposes to characterize transitions that *always* lead to an error in abstract models; however, this kind of approach requires enumerating the predicates and/or transitions; hence, it does not apply to ASTRÉE, due to the number of predicates in the abstract invariants (domains nearly infinite).

Debugging methods start with a *concrete* trace, which we precisely do not have, since alarms arise from abstract analyzes.

8.6.3 Perspectives

Currently, the implementation still requires a considerable amount of work in order to become really practical, even though we are able to propose early experimental results obtained with a prototype; the purpose of this short experiments was merely to assess whether this technique would provide some insightful results.

Moreover, we wish to investigate the automatic generation of semantic slicing criteria, and to use dependences results in order to assist it.

Last, another possible direction for future work would be to express abstract dependences involving more complicated, e.g. relational abstractions. Indeed, tracking the origin of an alarm raised in the analysis of $z = \sqrt{x + y}$ requires looking at dependences involving the property $x + y < 0$. This would require a much more general definition of dependences, so as to let dependences among predicates, and not just dependences among variables.

Part IV

Certified Compilation

Chapter 9

Formalizing Compilation

The two previous parts of this thesis aim at improving the precision of the analysis of source code (e.g., C programs). However, the certification of executable programs may require properties to be proved at the object code level, if the analysis of the source code cannot be considered a sufficient guarantee. Object code is usually produced by compiling source programs. Therefore, we envisage now the certification of compiled programs.

This chapter aims at describing our main motivations in certified compilation and at defining an adequate model for compilation, so that certification algorithms can be designed independently from the compiler we design them to for. The next two chapters describe two methods for certified compilation: invariant traduction and checking in Chapter 10, and translation validation (aka equivalence checking) in Chapter 11.

We detail the goal of these approaches to certified compilation in Section 9.1. We present the salient features of a simple, yet representative assembly language in Section 9.2. Section 9.3 formalizes the notion of non-optimizing compilation. We consider the case of optimizing compilation in Section 9.4.

9.1 Motivation

9.1.1 Certification of compiled code

Compilers are complex pieces of software; hence, we should expect them to potentially contain bugs. For instance, the Gnu C Compiler (**gcc**) amounts to more than 500 000 LOCs (Lines Of Code). Reports of bugs are rather frequent (and can be consulted on `http://gcc.gnu.org/ml/gcc-bugs/`). A compiler bug may have several consequences: crash of the compilation (which can be considered harmless, since it would not cause any severe damage), failure to comply with the semantics of the source language (e.g., wrong implementation of typing conversions, which may cause a fatal interruption to be raised at execution time), production of incorrect code (with many possible consequences, ranging from unexpected runtime errors to mis-implementation of critical functions of the source program). Obviously, the consequences of the production of incorrect code should be considered a very serious risk in the case of critical applications. The non-compliance with the semantics of the source language is also a serious issue: indeed, the analysis of source programs by analyzers like ASTRÉE is based on the semantics of the source language; hence, in case the compiler does not comply with the semantics, then one cannot consider the result of the analysis a proof for the safety of the executable program. Last, the very definition of some errors can be stated in a more easy way at the assembly level, as is the case of integer arithmetic operations.

In the following of this part, we will consider two approaches to certified compilation:

- **Invariant traduction** [Nec97, Riv03]: The goal of this method is to attempt to check that some abstract property of the source program also holds true for the compiled program. In particular, this approach allows to check that the executable code enjoys some safety property (e.g., the absence of runtime errors or the safety of memory operations). Therefore, it is a good way to get a good level of confidence in the safety of the program actually executed i.e., the assembly program instead of the source code.
- **Translation validation** [PSS98, Riv04b]: This approach proves the semantic equivalence of the source and the compiled program, using theorem proving methods. It allows to prove the functional correctness of the compiled program, i.e. that it implements correctly the functions implemented in the source program. It is also adapted to the documentation of the compilation, which is required by some development protocols [TCoA99].

Other approaches to certified compilation exist. In particular, we can cite theorem proving methods, which are based on a formal proof of the compiler: in case the compiler can be proved correct and the proof is trusted, then the functional equivalence of the source and compiled programs holds for any source program. Similarly, in case the source program is proved safe, the assembly program is safe. The downside of this solution is that it is often considered expensive (proving a compiler requires an important human effort) or not practical (in case the code of the compiler is not freely available or may be modified frequently). At the time we are writing this thesis, we can cite the proof of a mini-compiler in the Coq proof assistant [Ber98]. A more ambitious, ongoing project aims at proving a fully functional optimizing C compiler; no publication is currently available about this project, but information can be found at `http://www-sop.inria.fr/lemme/concert/`.

9.1.2 Formalizing compilation

We start this part with a formalization of compilation. The purpose of this approach is to define what should be meant by "compilation correctness" in a first step, before we state the compilation certification algorithms. The advantage of this approach is to make the certification algorithms as parametric as possible.

Indeed, a compiler may carry out the translation of programs in many different ways, and we would like to avoid algorithms or implementations of certification methods to be specific to a particular compiler or to a given architecture. In particular, we may point out the following issues:

- The translation of some structures (e.g., conditions, function calls) depends on the *architecture* and the Application Binary Interface (ABI) the code is compiled for.
- Most compilers attempt to produce *optimized* code, i.e. by reducing the size of the object code (number of instructions) or by making it faster. For instance, modern architectures allow several instructions to be executed in the same time thanks to instruction level parallelism, so as to speed up computations involving instructions that require several cycles to complete.

Therefore we start by giving a model of compilation (with or without optimizations) in this chapter, which should capture precisely the properties preserved by compilation transformations. The algorithms described in the following chapters will be based on the model given in this chapter.

The goal of this approach is to allow the reuse of these algorithms for a different compiler than the one chosen to assess them during their design, with a reduced amount of adaptations.

9.2 A Simple Assembly Language

First, we define a simple assembly language, derived from the Power-PC 32-bits assembly language, which was used for all the implementations carried out during this thesis. This processor features a rather symmetric RISC (Reduced Instruction Set Computing) architecture, so that the instruction set is rather simple to study.

9.2.1 Syntax

Memory cells: The architecture we consider features several kinds of memory locations: registers and memory cells. More precisely, we consider:

- **General-Purpose Registers** (for short, gpr): the n_{gpr} (in practice, $n_{\text{gpr}} = 32$) general-purpose registers are used for integer arithmetic and computations involving pointers; they are denoted with $\mathbf{gpr}_i$ (where $0 \leq i < n_{\text{gpr}}$);
- **Floating-Point Registers** (for short, fpr): the n_{fpr} (in practice, $n_{\text{fpr}} = 32$) floating-point registers are used for (32 and 64 bits) floating-point computations; they are denoted with $\mathbf{fpr}_i$ (where $0 \leq i < n_{\text{fpr}}$);
- **Condition Registers** (cr): the n_{cr} (in practice, $n_{\text{cr}} = 8$) condition registers store the result of conditions and determine the result of conditional branchings as well; they are denoted with $\mathbf{cr}_i$ (where $0 \leq i < n_{\text{cr}}$);
- **Memory cells**: they store the value of global or local variables. A memory cell is characterized with an integer address: we write $\mathbf{M}[\underline{d}]$ for the memory cell of address $\underline{d}$, where d is an integer.

Real architectures feature more registers. More precisely, one usually finds special registers for controlling the behavior of the processor regarding to exception handling, the behavior of floating-point operations (rounding mode, activation or deactivation of interruptions for overflows or underflows...), machine state, memory management (e.g., definition of active segments)... We restrict to the main registers in order to make the presentation more readable. Anyway, these special registers would be abstracted away when defining the correctness of compilation, in Section 9.3.

Values: General-purpose (resp. floating point) registers store integer (resp. floating point) values. Memory cells store fixed length bit-fields, which may be interpreted either as integers or as floating-point values.

Condition registers store values corresponding to the result of the evaluation of conditions: LT stands for "less than"; EQ stands for "equal" and GT stands for "greater than".

Control states are represented with program counter values (i.e., integers).

Instructions: We consider a very reduced kernel of the Power-PC assembly language:

- **arithmetic operations:** the classical 3-registers arithmetic instructions input two scalar values read in registers and store the result into a third register in case the computation succeeds; they cause the execution to crash otherwise (e.g., division by 0); such instructions are denoted with `op` $\mathbf{gpr}_i$, $\mathbf{gpr}_j$, $\mathbf{gpr}_k$ where `op` corresponds to the operation ($\mathtt{op} \in \{\mathtt{add}, \mathtt{mul}, \mathtt{fadd}, \ldots\}$);
- **load of constant value into a register:** the instruction `li` $\mathbf{gpr}_i$, v assigns the value v to register $\mathbf{gpr}_i$ (it also allows to load a constant value into a floating point register);
- **load from the memory:** if d is an integer and x is either an integer register or an integer value, then the instruction `load` $\mathbf{gpr}_i$, $\underline{d}\,(x)$ loads the content of the memory location of address $\underline{d} + x$ into the register $\mathbf{gpr}_i$, if it is a valid address; otherwise, it causes the execution of the program to crash due to a memory error; this instruction allows the access to scalar and compound type variables (this instruction works also for floating point registers);
- **store into the memory:** the instruction `store` $\mathbf{gpr}_i$, $\underline{d}\,(x)$ carries out the converse operation;
- **comparison:** the instruction `cmp` $\mathbf{cr}_i$, $\mathbf{gpr}_j$, $\mathbf{gpr}_k$ compares the values contained in registers $\mathbf{gpr}_j$ and $\mathbf{gpr}_k$ and stores the result into register $\mathbf{cr}_i$ (the same instruction is also defined for floating point registers): for instance, if the value in $\mathbf{gpr}_j$ is smaller than the value in $\mathbf{gpr}_k$, then, this instruction assigns the value LT to the condition register $\mathbf{cr}_i$;
- **branching:** if ℓ is a control state, the instruction `b` ℓ directs the execution to the instruction corresponding to line ℓ (by assigning the program counter);
- **conditional branching:** if c is a condition (i.e., condition register value) and ℓ a control state, then the instruction $\mathtt{bc}(c)$ $\mathbf{cr}_i, \ell$ branches to the instruction corresponding to line ℓ if the condition register $\mathbf{cr}_i$ contains a value equal to c; otherwise the execution continues at the next instruction.

memory location	notation	value
general-purpose register	$\mathbf{gpr}_i$, where $0 \leq i < n_{\text{gpr}}$	integer
floating-point register	$\mathbf{fpr}_i$, where $0 \leq i < n_{\text{fpr}}$	floating-point
condition register	$\mathbf{cr}_i$, where $0 \leq i < n_{\text{cr}}$	$\mathbb{C} = \{\text{LT}, \text{EQ}, \text{GT}\}$

(a) Memory locations and values

instruction	notation
arithmetic operations	$\texttt{op}\ \mathbf{gpr}_i,\ \mathbf{gpr}_j,\ \mathbf{gpr}_k$ where $\texttt{op} \in \{\texttt{add}, \texttt{mul}, \texttt{fadd}, \ldots\}$
load constant	$\texttt{li}\ \mathbf{gpr}_i,\ v$, where $v \in \mathbb{N}$
load from memory	$\texttt{load}\ \mathbf{gpr}_i,\ \underline{d}\,(x)$, where $d \in \mathbb{N}$ and x is an integer register or value
store into memory	$\texttt{store}\ \mathbf{gpr}_i,\ \underline{d}\,(x)$, where $d \in \mathbb{N}$ and x is an integer register or value
comparison	$\texttt{cmp}\ \mathbf{cr}_i,\ \mathbf{gpr}_j,\ \mathbf{gpr}_k$
branching	$\texttt{b}\ \ell$, where $\ell \in \mathbb{L}$
conditional branching	$\texttt{bc}(c)\ \mathbf{cr}_i, \ell$, where $c \in \mathbb{C}, \ell \in \mathbb{L}$

(b) Instruction set

Figure 9.1: A micro Power-PC assembly language

Other instructions may be introduced, when dealing with particular features of the processor.

Figure 9.1 summarizes the definition of the fragment of the Power-PC assembly language considered in this thesis.

9.2.2 Semantics

The semantics of the assembly language can be defined in a similar way as for the source language in Section 2.2.3:

- the assembly stores are completely defined by the sets of memory locations and corresponding values introduced in Section 9.2.1;
- the control states were also defined in Section 9.2.1 (a control state correspond to a value for the program counter);
- the definition of a set of states follows from the definitions of control and memory states;
- a transition relation defines what computation steps are feasible, for each instruction in the language.

We define the transition relation by the means of a family of symbolic transfer functions, as proposed in Section 3.2.6. Figure 9.2 defines the symbolic transfer functions corresponding to each instruction in the language.

More precisely, if ℓ is the control state right before an instruction, then $\mathbf{nxt}(\ell)$ denotes the control state right after the instruction (i.e., right before the next instruction). For each instruction, we give on Figure 9.2 the transfer function $\delta_{\ell,\mathbf{nxt}(\ell)}$ and any other transfer function corresponding to an edge which may be taken; if no symbolic transfer function is expressly defined for the one step transition between ℓ and ℓ', then this transfer function is $\delta_{\ell,\ell'} = \square$. Moreover, we use the following definitions:

- we write $\mathbf{is\_ok}(e_0 \oplus e_1)$ for the boolean expression which evaluates to **true** if the evaluation of the expression $e_0 \oplus e_1$ succeeds; it evaluates to **false** if the evaluation of $e_0 \oplus e_1$ results in an error;
- we write $\mathbf{is\_addr}(d)$ for a boolean expression which evaluates to **true** if the integer d denotes a valid address.

Our choice to resort to symbolic transfer functions for this definition is motivated by the fact that we will need to express the semantics of assembly programs along some finite paths, as defined in Section 3.2.3, so as to compare source and compiled programs. Symbolic transfer functions are precisely well adapted for this application.

9.3 Compilation

9.3.1 A simple example

In this section, we focus on non-optimizing compilation: we assume that the transformations performed by the compiler are simple and preserve the structure of programs (no interleaving of the compiled code for successive expressions, no global rewriting of the control structures). Our purpose is to define what properties of the source program is preserved by the compilation. More involved transformation will be considered in Section 9.4.

Let us look at the example given on Figure 9.3: on Figure 9.3(a), we show a source program, which computes the sum of the elements of an integer array t of length n; on Figure 9.3(b), we show a compiled version, with no optimization.

Clearly, this transformation is straightforward: the series of instructions corresponding to each instruction in the source code appear clearly as blocks of consecutive instructions, as summarized in the table below:

instruction	symbolic transfer function(s)
arithmetic operation `op` $\mathbf{gpr}_i$, $\mathbf{gpr}_j$, $\mathbf{gpr}_k$	$\delta_{\ell,\mathbf{nxt}(\ell)} = \begin{cases} \lfloor \mathbf{is\_ok}(\mathbf{gpr}_j \oplus \mathbf{gpr}_k) \ ?\ \lfloor \mathbf{gpr}_i \leftarrow \mathbf{gpr}_j \oplus \mathbf{gpr}_k \rfloor \\ \quad \mid \Box \rfloor \end{cases}$ where $\oplus$ is the operation corresponding to `op`
load constant `li` $\mathbf{gpr}_i$, v	$\delta_{\ell,\mathbf{nxt}(\ell)} = \lfloor \mathbf{gpr}_i \leftarrow v \rfloor$
load from memory `load` $\mathbf{gpr}_i$, $\underline{d}\,(x)$	$\delta_{\ell,\mathbf{nxt}(\ell)} = \lfloor \mathbf{is\_addr}(\underline{d} + x) \ ?\ \lfloor \mathbf{gpr}_i \leftarrow \mathbf{M}[\underline{d} + x] \rfloor \ \mid \ \Box \ \rfloor$
store into memory `store` $\mathbf{gpr}_i$, $\underline{d}\,(x)$	$\delta_{\ell,\mathbf{nxt}(\ell)} = \lfloor \mathbf{is\_addr}(\underline{d} + x) \ ?\ \lfloor \mathbf{M}[\underline{d} + x] \leftarrow \mathbf{gpr}_i \rfloor \ \mid \ \Box \ \rfloor$
comparison `cmp` $\mathbf{cr}_i$, $\mathbf{gpr}_j$, $\mathbf{gpr}_k$	$\delta_{\ell,\mathbf{nxt}(\ell)} = \begin{cases} \lfloor \mathbf{gpr}_j < \mathbf{gpr}_k \ ?\ \lfloor \mathbf{cr}_i \leftarrow \mathrm{LT} \rfloor \\ \quad \mid \ \lfloor \mathbf{gpr}_j = \mathbf{gpr}_k \ ?\ \lfloor \mathbf{cr}_i \leftarrow \mathrm{EQ} \rfloor \\ \qquad \mid \ \lfloor \mathbf{cr}_i \leftarrow \mathrm{GT} \rfloor \rfloor \rfloor \end{cases}$
branching `b` ℓ_b	$\delta_{\ell,\mathbf{nxt}(\ell)} = \Box$ $\delta_{\ell,\ell_b} = \iota$
conditional branching `bc(<)` $\mathbf{cr}_i$, ℓ_b	$\delta_{\ell,\mathbf{nxt}(\ell)} = \lfloor \mathbf{cr}_i = \mathrm{LT} \ ?\ \Box \ \mid \ \iota \ \rfloor$ $\delta_{\ell,\ell_b} = \lfloor \mathbf{cr}_i = \mathrm{LT} \ ?\ \iota \ \mid \ \Box \ \rfloor$

Figure 9.2: Symbolic transfer functions

```
$i, x$ :   integer variables
$t$ :      integer array of length $n \in \mathbb{N}$, where $n$ is a parameter

$\ell^s_0$    $i := -1;$
$\ell^s_1$    $x := 0;$
$\ell^s_2$    while($i < n$){
$\ell^s_3$        $i := i + 1;$
$\ell^s_4$        $x := x + t[i]$
$\ell^s_5$    }
$\ell^s_6$    ...
```

(a) Source program P_s

```
$\ell^c_0$   li      $\mathbf{gpr}_0$, $-1$                  $\ell^c_{10}$  add     $\mathbf{gpr}_0$, $\mathbf{gpr}_0$, $\mathbf{gpr}_1$
$\ell^c_1$   store   $\mathbf{gpr}_0$, $\underline{i}\,(0)$            $\ell^c_{11}$  store   $\mathbf{gpr}_0$, $\underline{i}\,(0)$
$\ell^c_2$   li      $\mathbf{gpr}_1$, $0$                   $\ell^c_{12}$  load    $\mathbf{gpr}_0$, $\underline{i}\,(0)$
$\ell^c_3$   store   $\mathbf{gpr}_1$, $\underline{x}\,(0)$            $\ell^c_{13}$  load    $\mathbf{gpr}_1$, $\underline{x}\,(0)$
$\ell^c_4$   load    $\mathbf{gpr}_0$, $\underline{i}\,(0)$            $\ell^c_{14}$  load    $\mathbf{gpr}_2$, $\underline{t}\,(\mathbf{gpr}_0)$
$\ell^c_5$   li      $\mathbf{gpr}_1$, $n$                   $\ell^c_{15}$  add     $\mathbf{gpr}_1$, $\mathbf{gpr}_1$, $\mathbf{gpr}_2$
$\ell^c_6$   cmp     $\mathbf{cr}_0$, $\mathbf{gpr}_0$, $\mathbf{gpr}_1$       $\ell^c_{16}$  store   $\mathbf{gpr}_1$, $\underline{x}\,(0)$
$\ell^c_7$   bc($\geq$) $\mathbf{cr}_0$, $\ell^c_{18}$                $\ell^c_{17}$  b       $\ell^c_4$
$\ell^c_8$   load    $\mathbf{gpr}_0$, $\underline{i}\,(0)$            $\ell^c_{18}$  ...
$\ell^c_9$   li      $\mathbf{gpr}_1$, $1$
```

(b) Assembly program P_c

Figure 9.3: Example compilation

instruction	series of assembly instructions (denoted with the corresponding program counter)
ℓ^s_0 : $i := -1;$	ℓ^c_0, ℓ^c_1
ℓ^s_1 : $x := 0;$	ℓ^c_2, ℓ^c_3
condition of the loop at l^s_2 and conditional branching	$\ell^c_4, \ell^c_5, \ell^c_6$ ℓ^c_7
ℓ^s_3 : $i := i + 1;$	$\ell^c_8, \ell^c_9, \ell^c_{10}, \ell^c_{11}$
ℓ^s_4 : $x := x + t[i]$	$\ell^c_{12}, \ldots, \ell^c_{16}$
loop back edge	ℓ^c_{17}
end of the program (ℓ^s_6)	ℓ^c_{18}

In particular, any computation corresponding to the assignments at ℓ^s_0 and ℓ^s_1 is finished before the code corresponding to the loop is executed. Therefore, we can relate precisely the state of the assembly program at ℓ^c_4 to the state of the source program at point ℓ^s_2.

In fact, we can establish a similar relation for any control state in the source program, as displayed on Figure 9.4(a). This mapping is defined formally as a function $\Pi_{\mathbb{L}}$, which maps control states in the source program into control states in the compiled program, according to the relation mentioned above.

In fact, a similar remark applies to memory locations. The content of the memory cell of address $\underline{x}$ corresponds to the value of variable x, whenever we reach a control state in correspondence, according to Figure 9.4(a). Therefore, we provide a mapping of memory locations on Figure 9.4(b). Again, this mapping $\Pi_{\mathbb{X}}$ is defined as a function, which maps source memory locations into assembly memory locations, according to the relation exhibited between the source and compiled programs.

$$\begin{array}{rlcl} \Pi_{\mathbb{L}}: & l_0^s & \mapsto & l_0^c \\ & l_1^s & \mapsto & l_2^c \\ & l_2^s & \mapsto & l_4^c \\ & l_3^s & \mapsto & l_8^c \\ & l_4^s & \mapsto & l_{12}^c \\ & l_5^s & \mapsto & l_{17}^c \\ & l_6^s & \mapsto & l_{18}^c \end{array}$$

(a) Control states mapping

$$\begin{array}{rlcl} \Pi_{\mathbb{X}}: & x & \mapsto & \mathbf{M}[\underline{x}] \\ & i & \mapsto & \mathbf{M}[\underline{i}] \\ & t[j] & \mapsto & \mathbf{M}[\underline{t}+j] \end{array}$$

(note: memory alignments are not taken into account in this example)

(b) Memory locations mapping

Figure 9.4: Mapping between source and compiled programs

Note that registers and "intermediate" control states (i.e., assembly control states in the middle of the blocks encoding source instructions) do not appear in the mappings displayed in Figure 9.4, since they do not have a counterpart in the source program.

9.3.2 Abstraction

The previous subsection showed a simple example of compilation and showed what control states and memory locations of both programs could be related. Therefore, we now provide a formalization of compilation, using the scheme given in Section 2.3.4, so as to describe what is meant by "correct compilation".

In particular, Section 9.3.1 shows that some control states or memory states of the assembly program cannot be related with anything in the source program. This suggests using the projection abstraction introduced in Section 3.4, so as to remove them: the semantics of the source program can be related to an abstraction of the assembly program, defined by a subset of control states and memory locations. Moreover, a compiler may remove some control states and variables of the source program, for instance, if it carries out some kind of constant propagation and/or dead-code elimination (such as the example given in Section 3.4).

Therefore, we define restricted sets of control states and memory locations, as in Section 3.4.1 and Section 3.4.2. We write $\mathbb{X}_{\mathrm{s}}$ (resp. $\mathbb{X}_{\mathrm{c}}$) for the memory locations of the source (resp. compiled) program, and $\mathbb{L}_{\mathrm{s}}$ (resp. $\mathbb{L}_{\mathrm{c}}$) for the set of control states of the source (resp. compiled) program. Moreover, we introduce the following restricted sets:

- for the **source program:** $\overline{\mathbb{X}}_{\mathrm{s}} \subseteq \mathbb{X}_{\mathrm{s}}$ and $\overline{\mathbb{L}}_{\mathrm{s}} \subseteq \mathbb{L}_{\mathrm{s}}$;
- for the **assembly program:** $\overline{\mathbb{X}}_{\mathrm{c}} \subseteq \mathbb{X}_{\mathrm{c}}$ and $\overline{\mathbb{L}}_{\mathrm{c}} \subseteq \mathbb{L}_{\mathrm{c}}$.

Moreover, we extend the notations for states and for traces to the source and compiled programs accordingly: $._{\mathrm{s}}$ denotes an object of the source program; $._{\mathrm{c}}$ denotes an object of the compiled program; $\bar{\cdot}$ denotes a restricted set, as in Section 3.4. For instance, we write $\overline{\Sigma}_{\mathrm{s}}$ denotes the set of restricted traces for the source program.

Let $\Pi_{\mathbb{X}} : \overline{\mathbb{X}}_{\mathrm{s}} \longrightarrow \overline{\mathbb{X}}_{\mathrm{c}}$ and $\Pi_{\mathbb{L}} : \overline{\mathbb{L}}_{\mathrm{s}} \longrightarrow \overline{\mathbb{L}}_{\mathrm{c}}$ be two bijections, defined in the same way as in Section 9.3.1. We let the store mapping $\Pi_{\mathbb{M}} : \overline{\mathbb{M}}_{\mathrm{s}} \longrightarrow \overline{\mathbb{M}}_{\mathrm{c}}$ be defined by $\Pi_{\mathbb{M}}(\rho) = \lambda(x \in \overline{\mathbb{X}}_{\mathrm{c}}) \cdot \rho((\Pi_{\mathbb{X}})^{-1}(x))$. Moreover, we define Π_{Σ} by:

$$\begin{array}{llcl} \Pi_{\Sigma}: & \overline{\Sigma}_{\mathrm{s}} & \longrightarrow & \overline{\Sigma}_{\mathrm{c}} \\ & \langle(\ell_0,\rho_0),\ldots,(\ell_n,\rho_n)\rangle & \longmapsto & \langle(\Pi_{\mathbb{L}}(\ell_0),\Pi_{\mathbb{X}}(\rho_0)),\ldots(\Pi_{\mathbb{L}}(\ell_n),\Pi_{\mathbb{X}}(\rho_n))\rangle \end{array}$$

Definition 9.3.1. Correctness of compilation.

We say that the compilation of P_{s} *into* P_{c} *is* Π_{Σ}-correct *if and only if* Π_{Σ} *is a bijection between the projected traces of the source and of the compiled program:*

$$\alpha_{\Pi\langle\overline{\mathbb{X}}_{\mathrm{s}},\overline{\mathbb{L}}_{\mathrm{s}}\rangle}(\llbracket P_{\mathrm{s}}\rrbracket) \overset{\Pi_{\Sigma}}{\simeq} \alpha_{\Pi\langle\overline{\mathbb{X}}_{\mathrm{c}},\overline{\mathbb{L}}_{\mathrm{c}}\rangle}(\llbracket P_{\mathrm{c}}\rrbracket)$$

This situation can be described in the diagram below, similar to the one in Section 2.3.4.

$$\begin{tikzcd}
P_s \arrow[rr, mapsto, "\textit{compilation}"] \arrow[d, mapsto, "\text{semantics}"'] & & P_c \arrow[d, mapsto, "\text{semantics}"] \\
\llbracket P_s\rrbracket \arrow[d, "\alpha_{\Pi\langle\overline{\mathbb{X}}_{\mathrm{s}},\overline{\mathbb{L}}_{\mathrm{s}}\rangle}"] & & \llbracket P_c\rrbracket \arrow[d, "\alpha_{\Pi\langle\overline{\mathbb{X}}_{\mathrm{c}},\overline{\mathbb{L}}_{\mathrm{c}}\rangle}"] \\
\alpha_{\Pi\langle\overline{\mathbb{X}}_{\mathrm{s}},\overline{\mathbb{L}}_{\mathrm{s}}\rangle}(\llbracket P_s\rrbracket) \arrow[rr, mapsto, "\Pi_{\Sigma}"'] & & \alpha_{\Pi\langle\overline{\mathbb{X}}_{\mathrm{c}},\overline{\mathbb{L}}_{\mathrm{c}}\rangle}(\llbracket P_c\rrbracket)
\end{tikzcd}$$

Intuitively, the correctness of the compilation of P_s into P_c states that an execution of P_c corresponds to an execution of P_s up-to some bijection and reciprocally.

Definition 9.3.1. Projections.

In particular, in the example of Section 9.3.1,

- $\overline{\mathbb{X}}_{\rm s} = \mathbb{X}_{\rm s}$ *and* $\overline{\mathbb{L}}_{\rm s} = \mathbb{L}_{\rm s}$*;*
- $\overline{\mathbb{X}}_{\rm c} = \{\mathbf{M}[\underline{x}], \mathbf{M}[\underline{i}]\} \cup \{\mathbf{M}[\underline{t} + j] \mid j \in (\!|0, n-1|\!)\}$*;*
- $\overline{\mathbb{L}}_{\rm c} = \{\ell_0^c, \ell_2^c, \ell_4^c, \ell_8^c, \ell_{12}^c, \ell_{17}^c, \ell_{18}^c\}$

As a consequence, the compilation of $P_{\rm s}$ into $P_{\rm c}$ (Figure 9.3) is correct in the sense of Definition 9.3.1.

This statement can be compared to what could be expressed using bisimulation methods [Mil90]. However, we stress the importance of the projection abstractions involved in the definition of the correctness of compilation. Indeed, the generalization to some basic optimizations in Section 9.4 will mainly be based on a tuning of these abstractions. Moreover, these abstractions allow to define what the compilation preserves, i.e. a kind of *invariant* for the transformation.

Remark 9.3.1. Dealing with scopes.
Most of the time, variables have a restricted scope: for instance, local variables are only relevant in a block of code or in a function. Therefore, the set of memory locations depends on the control state. As a result, the mapping of memory locations $\Pi_{\mathbb{X}}$ should depend on the control state: it should be defined as a function $\Pi_{\mathbb{X}} : \overline{\mathbb{L}}_{\rm s} \times \overline{\mathbb{X}}_{\rm s} \rightarrow \overline{\mathbb{X}}_{\rm c}$, such that $\Pi_{\mathbb{X}}(\ell, x)$ is the assembly memory location corresponding to x at point ℓ.

9.3.3 Reduced program

We now propose to provide a least-fixpoint definition for the projected semantics, defined in Section 9.3.2. Basically, we propose to give a constructive version of the result given in Section 3.4.4, Lemma 3.4.1, by defining a "program reduction" technique, allowing to replace an assembly program with another program, which is equivalent modulo the abstraction defined in Section 9.3.2.

First, we make a few assumptions:

- we consider here the case of the assembly program only, i.e. we assume $\overline{\mathbb{X}}_{\rm s} = \mathbb{X}_{\rm s}$ and $\overline{\mathbb{L}}_{\rm s} = \mathbb{L}$ (the compiler does not remove any part of the program): the technique explained below would also apply to the source program;
- we assume that $\Pi_{\mathbb{L}}(\ell_{\rm s}^{\rm i}) = \ell_{\rm c}^{\rm i}$, i.e., the entry point of the source program corresponds to the entry point of the assembly program;
- we assume that the compiler does not insert a loop in the assembly program, which does not correspond to a loop in the source program, so that any loop in the compiled code corresponds to a loop in the source code.

The first assumption is made so as to keep the presentation short; the latter two hypotheses are very reasonable (we expect any compiler to satisfy them).

As a consequence of the second assumption, any loop in the compiled program P_c contains at least one point in $\overline{\mathbb{L}}_{\rm c}$.

The principle of program reduction is to define transitions corresponding to several steps in P_c, between control states in $\overline{\mathbb{L}}_{\rm c}$:

Definition 9.3.2. Reduced program.

The reduced program $P_{\mathrm{c}}^{\mathrm{r}}$ is defined as follows:

- *the set of control states is $\overline{\mathbb{L}}_{\mathrm{c}}$;*
- *the initial control state is $l_{\mathrm{c}}^{\mathrm{i}}$;*
- *the transition relation is defined by a family of symbolic transfer functions derived from the symbolic transfer functions of P_{c} by composition along sets of paths: if $l_{\vdash}, l_{\dashv} \in \overline{\mathbb{L}}_{\mathrm{c}}$, then $\delta_{l_{\vdash},l_{\dashv}}$ is the symbolic representation the denotational semantics corresponding to the set set of paths of the form $p = l_{\vdash}\cdot l_0 \cdot \ldots \cdot l_n \cdot l_{\dashv}$, $\forall i \in (\!|0, n|\!)$, $l_i \notin \overline{\mathbb{L}}_{\mathrm{c}}$ (the symbolic representation for a set of paths was defined in Lemma 3.2.6).*

In practice, the computation of the reduced program relies on the composition operation $\oplus$ (Section 3.2.6), and possibly on some simplification operation *simplify*. The advantages inherent in the use of a simplification function at this point will be stated in the following chapters (i.e., they appear at verification time).

Compiler ofter split paths for conditions: for instance, the branching corresponding to a condition like $e_0 \vee e_1 \vee e_2$ may be split in several branchings, so as to not to evaluate e_1, e_2 if e_0 is true. Should that case arise, Lemma 3.2.6 provides an algorithm to associate a single symbolic transfer function to the resulting set of paths.

Moreover, the computation of the reduced program requires the restricted sets of control states and memory locations, to be known. In practice the compilers provide debugging information (such as Stabs or Dwarf formats), including mappings $\Pi_{\mathbb{L}}, \Pi_{\mathbb{X}}$, which allow to define the restricted sets. Some algorithms were proposed so as to recover these mappings, when the compiler does not provide these information, e.g. in [TG00b, TG00a].

Definition 9.3.2. Projection of control states.

For instance, let us we consider the assembly program in Figure 9.3(b), with the restricted sets defined in Example 9.3.1. Then, the table of symbolic transfer functions for the reduced program $P_{\mathrm{c}}^{\mathrm{r}}$ is defined as follows:

$$
\begin{array}{rcl}
\delta_{l_0^c, l_2^c} & = & \lfloor \mathbf{gpr}_0 \leftarrow -1,\ \mathbf{M}[\underline{i}] \leftarrow -1 \rfloor \\
\delta_{l_2^c, l_4^c} & = & \lfloor \mathbf{gpr}_1 \leftarrow 0,\ \mathbf{M}[\underline{x}] \leftarrow 0 \rfloor \\
\delta_{l_4^c, l_8^c} & = & \left\{ \begin{array}{l} \lfloor \mathbf{M}[\underline{i}] < n\ ?\ \lfloor \mathbf{cr}_0 \leftarrow \mathrm{LT},\ \mathbf{gpr}_0 \leftarrow \mathbf{M}[\underline{i}],\ \mathbf{gpr}_1 \leftarrow n \rfloor \\ \qquad \mid\ \lfloor \mathbf{M}[\underline{i}] = n\ ?\ \square \\ \qquad\qquad \mid\ \square \rfloor \rfloor \end{array} \right. \\
\delta_{l_4^c, l_{18}^c} & = & \left\{ \begin{array}{l} \lfloor \mathbf{M}[\underline{i}] < n\ ?\ \square \\ \qquad \mid\ \lfloor \mathbf{M}[\underline{i}] = n\ ?\ \lfloor \mathbf{cr}_0 \leftarrow \mathrm{EQ},\ \mathbf{gpr}_0 \leftarrow \mathbf{M}[\underline{i}],\ \mathbf{gpr}_1 \leftarrow n \rfloor \\ \qquad\qquad \mid\ \lfloor \mathbf{cr}_0 \leftarrow \mathrm{GT},\ \mathbf{gpr}_0 \leftarrow \mathbf{M}[\underline{i}],\ \mathbf{gpr}_1 \leftarrow n \rfloor \rfloor \rfloor \end{array} \right. \\
\delta_{l_8^c, l_{12}^c} & = & \lfloor \mathbf{gpr}_0 \leftarrow \mathbf{M}[\underline{i}] + 1,\ \mathbf{gpr}_1 \leftarrow 1,\ \mathbf{M}[\underline{i}] \leftarrow \mathbf{M}[\underline{i}] + 1 \rfloor \\
\delta_{l_{12}^c, l_{17}^c} & = & \left\{ \begin{array}{l} \lfloor \mathbf{gpr}_0 \leftarrow \mathbf{M}[\underline{i}],\ \mathbf{gpr}_1 \leftarrow \mathbf{M}[\underline{x}], \\ \mathbf{gpr}_2 \leftarrow \mathbf{M}[\underline{t} + \mathbf{M}[\underline{i}]],\ \mathbf{M}[\underline{x}] \leftarrow \mathbf{M}[\underline{x}] + \mathbf{M}[\underline{t} + \mathbf{M}[\underline{i}]] \rfloor \end{array} \right. \\
\delta_{l_{17}^c, l_4^c} & = & \iota
\end{array}
$$

The soundness and completeness of this transformation with respect to the projection of the operational semantics writes down as follows:

Theorem 9.3.1. Adequation.

$$[\![P_{\mathrm{c}}^{\mathrm{r}}]\!] = \alpha_{\Pi\langle\bar{\mathbb{L}}_{\mathrm{c}}\rangle}([\![P_{\mathrm{c}}]\!])$$

Proof.
By induction on the length of traces. □

This definition of reduced programs focuses on the elimination of control states only; the elimination of the memory locations we would like to abstract away (such as the registers) can be carried out as a second step, by erasing these from the transfer functions of the reduced program:

Definition 9.3.3. Projection of memory locations.
For instance, let us we consider the assembly program of Figure 9.3(b), with the restricted sets defined in Example 9.3.1. Then, the table of symbolic transfer functions for the reduced program $P_{\mathrm{c}}^{\mathrm{r}}$ is defined as follows:

$$\begin{array}{rcl}
\delta_{l_0^c,l_2^c} & = & \lfloor \mathbf{M}[\underline{i}] \leftarrow -1 \rfloor \\
\delta_{l_2^c,l_4^c} & = & \lfloor \mathbf{M}[\underline{x}] \leftarrow 0 \rfloor \\
\delta_{l_4^c,l_8^c} & = & \lfloor \mathbf{M}[\underline{i}] < n\ ?\ \iota \mid \lfloor \mathbf{M}[\underline{i}] = n\ ?\ \Box \mid \Box \rfloor \rfloor \\
\delta_{l_4^c,l_{16}^c} & = & \lfloor \mathbf{M}[\underline{i}] < n\ ?\ \Box \mid \lfloor \mathbf{M}[\underline{i}] = n\ ?\ \iota \mid \iota \rfloor \rfloor \\
\delta_{l_8^c,l_{11}^c} & = & \lfloor \mathbf{M}[\underline{i}] \leftarrow \mathbf{M}[\underline{i}] + 1 \rfloor \\
\delta_{l_{11}^c,l_{15}^c} & = & \lfloor \mathbf{M}[\underline{x}] \leftarrow \mathbf{M}[\underline{x}] + \mathbf{M}[\underline{t} + \mathbf{M}[\underline{i}]] \rfloor \\
\delta_{l_{15}^c,l_4^c} & = & \iota
\end{array}$$

We can remark that in the above example, the symbolic transfer functions for the compiled program correspond exactly to the symbolic transfer functions for the source program up to the mapping for memory locations $\Pi_{\mathbb{X}}$, which we give below:

Definition 9.3.4. Source program.
The non-void symbolic transfer functions for the source program P_{s} given in Figure 9.3(a) are the following:

$$\begin{array}{rcl}
\delta_{l_0^s,l_1^s} & = & \lfloor i \leftarrow -1 \rfloor \\
\delta_{l_1^s,l_2^s} & = & \lfloor x \leftarrow 0 \rfloor \\
\delta_{l_2^s,l_3^s} & = & \lfloor i < n\ ?\ \iota \mid \Box \rfloor \\
\delta_{l_2^s,l_6^s} & = & \lfloor i < n\ ?\ \Box \mid \iota \rfloor \\
\delta_{l_3^s,l_4^s} & = & \lfloor i \leftarrow i + 1 \rfloor \\
\delta_{l_4^s,l_5^s} & = & \lfloor x \leftarrow x + t[i] \rfloor \\
\delta_{l_5^s,l_2^s} & = & \iota
\end{array}$$

In case some parts of the source program are removed and cannot be related to the compiled program, the same program reduction technique can be applied to the source program. At this point, we can state the definition of the correctness of compilation in terms of the reduction of the source and compiled programs:

Definition 9.3.3. Correctness of compilation, in terms of reduced programs. *Let P_{s} be a source program, compiled into P_{c}. We write $P_{\mathrm{s}}^{\mathrm{r}}$ and $P_{\mathrm{c}}^{\mathrm{r}}$ for the reduced programs and Π_{Σ} for the trace mapping defined by the mappings $\Pi_{\mathbb{L}}$ and $\Pi_{\mathbb{X}}$. Then, the compilation is* Π_{Σ}-correct *if and only if Π_{Σ} is a bijection between the semantics of the restricted source program and the semantics of the restricted compiled program:*

$$\llbracket P_{\mathrm{s}}^{\mathrm{r}} \rrbracket \overset{\Pi_{\Sigma}}{\simeq} \llbracket P_{\mathrm{c}}^{\mathrm{r}} \rrbracket$$

We have to prove that this definition is equivalent to our previous definition for compilation correctness (Definition 9.3.1):
Proof.
This statement of the correctness of compilation is equivalent to Definition 9.3.1, since the adequation of program reduction (Theorem 9.3.1) implies the equality $\llbracket P_{\mathrm{c}}^{\mathrm{r}} \rrbracket = \alpha_{\Pi\langle \overline{\mathbb{X}}_{\mathrm{c}}, \overline{\mathbb{L}}_{\mathrm{c}} \rangle}(\llbracket P_{\mathrm{c}} \rrbracket)$, and similarly for the compiled program. □

9.3.4 Compilation of function calls

We described a procedural extension of our simple source language in Section 2.2.4.

A procedural extension of the assembly language of Section 9.2 would be rather similar, except that it would represent a stack inside the memory, so as to record where the function return should branch to. By contrast, in Section 2.2.4, the stack is a mere extension of the control states. As a consequence, the main issue with the formulation of compilation correctness in presence of procedure is to map the state of the physical representation of the stack with the “syntactic” stack κ used in Section 2.2.4.

Last, it is a common practice to have the local variables stored in the stack, which makes the definition of the $\Pi_{\mathbb{X}}$ function slightly more involved.

9.4 Common Optimizations

9.4.1 How to cope optimizations ?

In the previous section, we assumed the compiler produces rather simple code, i.e. does not try to improve the code generated and just translates instructions in a *separate* (the object code for consecutive statements does not overlap), *context insensitive* way (a same source statement is translated in the same way, whatever the place it occurs in the program). This assumption usually is not valid. Even simple compilers attempt to increase the efficiency of the code they produce, e.g. by avoiding to store useless variables. Real compilers carry out much more ambitious transformations, by changing the order of instructions (e.g., instruction level parallelism) or deeply modifying execution paths (e.g., loop optimizations, such as loop unrolling). For a comprehensive introduction to compiler optimizations, we refer the reader to classical compilation books, such as [App99, WM94] or to the survey [BGS94].

The main issue with optimizations is that they tend to break the correspondences we set up in Section 9.3.2; as a consequence, the definition of compilation correctness given in Section 9.3.2 is broken, and so is the notion of reduced program introduced in Section 9.3.3. Not only the definition we stated previously would fail, but deciding what variables or control states of the source and compiled program should be related may not be obvious.

Optimizations usually either simplify the structure of compiled programs or re-organize the structure of the code, so as to improve performances. Therefore, we propose to adapt the definition of compilation correctness so as to consider such simplifications or re-organizations correct compilation. The new, extended definitions are based mainly on a careful extension of the program abstraction technique introduced in Section 9.3.2 and of more general algorithms for program reduction.

In the following, we consider the case of a series of representative optimizations and apply this methodology.

9.4.2 Code simplification

One of the most simple optimizations a compiler may carry out is the removal of dead code and of dead variables.

Constant propagation and dead-code elimination: Most compilers do a constant propagation analysis [Kil73], so as to remove constant variables, constant assignments, and evaluate constant conditions. We showed how this transformation is formalized inside the abstract interpretation framework in Section 3.4, by defining a projection of the semantics of the source program. Therefore, this transformation fits in our initial framework, since we based the definition for the correctness of compilation on an abstraction of the source program, defined by restriction of the source control states ($\overline{\mathbb{L}}_{\mathrm{s}}$) and memory locations ($\overline{\mathbb{X}}_{\mathrm{s}}$): we simply need to abstract away the control states corresponding to the instructions removed by the transformation.

Removal of dead-variables: In case a variable is not used anymore after some point, the compiler may remove it from the memory, so as to reduce memory usage. Then, such a variable cannot be related to any memory location after some point in the assembly program. This transformation is handled by a relational mapping $\Pi_{\mathbb{X}} : \overline{\mathbb{L}}_{\mathrm{s}} \times \overline{\mathbb{X}}_{\mathrm{s}} \to \overline{\mathbb{X}}_{\mathrm{c}}$, akin to the solution proposed in Remark 9.3.1. Indeed, this definition for memory location mappings allows to discard a variable at any point in the program.

Copy propagation and register coalescing: Compilers attempt to keep a variable that is used several times in a piece of code in a register and not to store it back into the memory before using it again. Again, we need to twick the mapping for memory locations. More precisely, we request $\Pi_{\mathbb{X}}$ to map a pair (ℓ, x) into a *set* of assembly memory locations, which store the same value as x at point l.

The following example illustrates this solution:

Definition 9.4.1. Register coalescing.
In the body of the loop of the source program P_{s} displayed in Figure 9.3(a), variable i is used several times, therefore it may be stored into a register. This would amount to replace the body of the loop with the following piece of code, where the instruction corresponding to ℓ^c_{12} is removed (smaller optimized code, hence faster execution):

$$\begin{array}{llllll}
\ell^c_8 & \texttt{load} & \mathbf{gpr}_0,\ \underline{i}\,(0) & \ell^c_{13} & \texttt{load} & \mathbf{gpr}_1,\ \underline{x}\,(0) \\
\ell^c_9 & \texttt{li} & \mathbf{gpr}_1,\ 1 & \ell^c_{14} & \texttt{load} & \mathbf{gpr}_2,\ \underline{t}\,(\mathbf{gpr}_0) \\
\ell^c_{10} & \texttt{add} & \mathbf{gpr}_0,\ \mathbf{gpr}_0,\ \mathbf{gpr}_1 & \ell^c_{15} & \texttt{add} & \mathbf{gpr}_1,\ \mathbf{gpr}_1,\ \mathbf{gpr}_2 \\
\ell^c_{11} & \texttt{store} & \mathbf{gpr}_0,\ \underline{i}\,(0) & \ell^c_{16} & \texttt{store} & \mathbf{gpr}_1,\ \underline{x}\,(0)
\end{array}$$

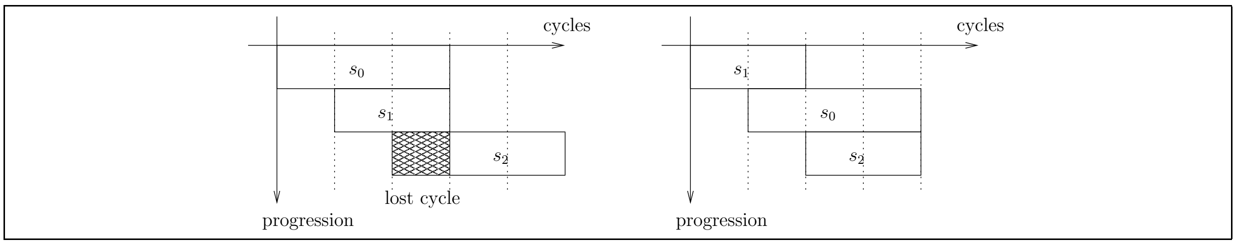
Then, after point ℓ^c_{11}, i corresponds to both $\mathbf{gpr}_0$ and $\mathbf{M}[\underline{i}]$. Hence, we would let:

$$\Pi_{\mathbb{X}}(\ell^s_4, i) = \Pi_{\mathbb{X}}(\ell^s_5, i) = \{\mathbf{gpr}_0, \mathbf{M}[\underline{i}]\}$$

9.4.3 Instruction level parallelism (scheduling)

We envisage now the case of a transformations which compromise the correspondence of program points; our study focuses on instruction scheduling. Instruction Level Parallelism (ILP or scheduling) aims at using the ability of executing several instructions simultaneously featured by modern architectures, so as to cut down the cost of several cycles long instructions. The number of cycles lost in the execution of an instruction is called the latency: a latency of one means that the execution of an instruction lasts two cycles instead of one. Several kinds of scheduling should be distinguished: hardware scheduling is implemented in the processor, which performs an ordering of instruction at run-time; the correctness of hardware scheduling is part of the specification of the processor, so its verification is beyond the scope of this thesis. Hence, it is somewhat part of the processor specification. By contrast, software scheduling is performed at compile time. Of course, we focus here on software scheduling.

A detailed introduction to software scheduling can be found in [App99], Chapter 20. The principle of software scheduling is to re-order the instructions of the compiled code, so as to allow independent tasks to be performed in the same time. For instance, if we consider a piece of code $s_0; s_1; s_2$ made of three instructions, such that s_1 and s_2 do not depend on s_0 but s_2 depends on the result of s_1: then, performing s_1 before s_0 does not change the behavior of the program, and may allow the execution of s_2 to be started faster. Therefore, the re-ordered code $s_1; s_0; s_2$ would produce a similar result and may yield better performances. The diagram below illustrate this fact; obviously, s_2 can start earlier in the case of the "re-scheduled" code.



Obviously, software scheduling does not fit in the correctness definition presented in Section 9.3.2, since the pieces of code corresponding to distinct source statements might be inter-wound, due to assembly instructions being permuted, which prevents from defining a mapping $\Pi_{\mathbb{L}}$, as shown in the example below.

Definition 9.4.2. Software scheduling.

We assume that all instructions have a latency of 1*, which is not completely realistic: usually memory instructions have a longer latency due to slower chips being accessed, whereas arithmetic instructions have no latency, since they are performed by a specialized unit inside the processor. In fact, the latency of a memory instruction depends on many parameters, including the layout of the cache. Our assumption is made for the sake of the simplicity of the example only.*
We consider the two pieces of code in Figure 9.5. The non-optimized code displayed in Figure 9.5(a) corresponds to the result of the register coalescing optimization presented in Example 9.4.1. The execution of this piece of 8 *instructions lasts* 12 *cycles: for instance, the execution of the load instruction at ℓ_1^n should complete before the addition at ℓ_2^n can be performed. Figure 9.5(b) presents an optimized version of this program, so as to cut down the number of stall cycles to* 1*: The mapping $\Pi_{\mathbb{L}}$ relates the source control state ℓ_4^s with ℓ_3^n. However, this mapping can no longer be defined in the case of the optimized program. Indeed, the value of x at ℓ_4^s corresponds to the value of $\mathbf{M}[\underline{x}]$ at ℓ_3^o (where it is copied into a register by a load instruction), whereas the value of i corresponds to the value of $\mathbf{M}[\underline{i}]$ at ℓ_7^o (i.e., after the new value is written into the memory).*

As illustrated in the example above, the difficulty in the definition of $\Pi_{\mathbb{X}}$ stems from the fact that the value of two source memory locations x_0, x_1 may correspond to the values of assembly memory locations at *distinct* control states in the optimized code.

As a consequence, we need to give a relaxed definition for the mappings $\Pi_{\mathbb{L}}$ and $\Pi_{\mathbb{X}}$, allowing to map a single control state $\ell^s \in \mathbb{L}_s$ of the source program into a series of control

ℓ_0^n	`li`	$\mathbf{gpr}_1$, 1
ℓ_1^n	`load`	$\mathbf{gpr}_0$, $\underline{i}\,(0)$
ℓ_2^n	`add`	$\mathbf{gpr}_0$, $\mathbf{gpr}_0$, $\mathbf{gpr}_1$
ℓ_3^n	`store`	$\mathbf{gpr}_0$, $\underline{i}\,(0)$
ℓ_4^n	`load`	$\mathbf{gpr}_1$, $\underline{x}\,(0)$
ℓ_5^n	`load`	$\mathbf{gpr}_2$, $\underline{t}\,(\mathbf{gpr}_0)$
ℓ_6^n	`add`	$\mathbf{gpr}_1$, $\mathbf{gpr}_1$, $\mathbf{gpr}_2$
ℓ_7^n	`store`	$\mathbf{gpr}_1$, $\underline{x}\,(0)$
ℓ_8^n	...	

(a) Non-optimized code

ℓ_0^o	`load`	$\mathbf{gpr}_0$, $\underline{i}\,(0)$
ℓ_1^o	`li`	$\mathbf{gpr}_1$, 1
ℓ_2^o	`add`	$\mathbf{gpr}_0$, $\mathbf{gpr}_0$, $\mathbf{gpr}_1$
ℓ_3^o	`load`	$\mathbf{gpr}_1$, $\underline{x}\,(0)$
ℓ_4^o	`load`	$\mathbf{gpr}_2$, $\underline{t}\,(\mathbf{gpr}_0)$
ℓ_5^o	`store`	$\mathbf{gpr}_0$, $\underline{i}\,(0)$
ℓ_6^o	`add`	$\mathbf{gpr}_1$, $\mathbf{gpr}_1$, $\mathbf{gpr}_2$
ℓ_7^o	`store`	$\mathbf{gpr}_1$, $\underline{x}\,(0)$
ℓ_8^o	...	

(b) Optimized code

Figure 9.5: Software scheduling

states $\ell_0^c, \ldots, \ell_n^c$ in the compiled code and to map a source memory location x^s into a tuple made of a memory location x_c of the compiled program and a control state ℓ_i^c chosen in the series $\ell_0^c, \ldots, \ell_n^c$.

Such a series of assembly control states is called a *fictitious control state*; we introduce this notion together with the corresponding definitions for $\Pi_{\mathbb{L}}$ and $\Pi_{\mathbb{X}}$:

Definition 9.4.1. Fictitious control state, fictitious state.

In case $\ell^s \in \mathbb{L}_s$ corresponds to the sequence of assembly control states $\ell_0^c, \ldots, \ell_n^c$, we introduce a fictitious label *ℓ^f representing this sequence and a set of* fictitious memory locations *$X_{\ell^f} \subseteq (\{\ell_0^c, \ldots, \ell_n^c\} \times \mathbb{X}_c)$: the couple (x, ℓ_i^c) represents the memory location x, and states that it should be observed at point ℓ_i^c. Furthermore, we assert that $\Pi_{\mathbb{L}}(l^s) = l^f$. Let $\langle(\ell_0^c, \rho_0^c), \ldots, (\ell_n^c, \rho_n^c)\rangle$ be a sequence of states corresponding to the above sequence of control states. We project them into a* fictitious state *(l^f, ρ^f), where ρ^f is defined by:*

$$\forall(\ell_i^c, x_c) \in X_{\ell^f},\ \rho^f(x_c) = \rho_i^c(x_c)$$

Then, $\Pi_{\mathbb{X}}(x_s) = x_c$ means that the value of x_s corresponds to the value of x_c at a point ℓ_i^c, such that $(\ell_i^c, x_c) \in X_{\ell^f}$.

We illustrate this notion in the case of the optimized code presented in Example 9.4.2:

Definition 9.4.3. Example 9.4.2 continued.

We let ℓ^f be the fictitious control state corresponding to ℓ_3^s in the optimized program displayed in Figure 9.5(b), and define the fictitious state as follows:

- *ℓ^f stands for the sequence $\ell_2^o, \ell_3^o, \ell_4^o, \ell_5^o, \ell_6^o, \ell_7^o$;*
- *the set of fictitious memory locations X_{ℓ^f} and the mapping $\Pi_{\mathbb{X}}$ are defined by:*
  - *$\Pi_{\mathbb{X}}(i) = \{\mathbf{M}[\underline{i}]\}$ and $\mathbf{M}[\underline{i}]$ is observed at ℓ_7^o: $(\ell_7^o, \mathbf{M}[\underline{i}]) \in X_{\ell^f}$;*

– $\Pi_{\mathbb{X}}(x) = \{\mathbf{M}[\underline{x}]\}$ *and* $\mathbf{M}[\underline{x}]$ *is observed at* ι_2^o*:* $(\iota_2^o, \mathbf{M}[\underline{x}]) \in X_{\iota^f}$*;*

– *the values for* t *are not modified and may be observed at any point in* $\iota_2^o, \dots, \iota_7^o$*.*

The situation is illustrated in the Figure 9.6; it shows what point variables x *and* i *should be observed at.*

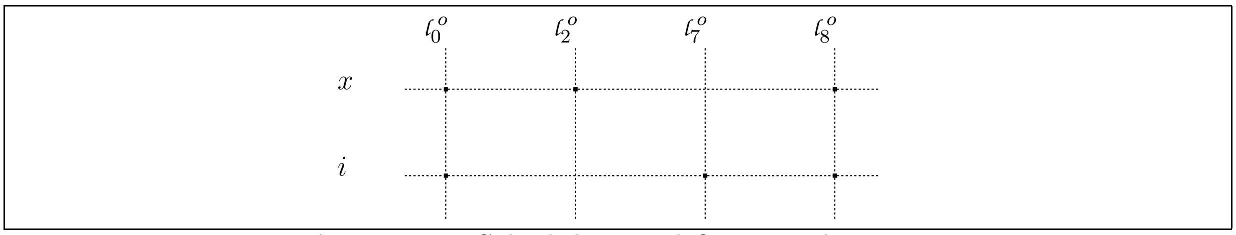


Figure 9.6: Scheduling and fictitious locations

The last issue is the computation of the reduced compiled program. Obviously, the algorithms presented in Section 9.3.3 need to be generalized. The new algorithms proceeds by composing partial symbolic transfer functions, representing the modification of the fictitious memory locations instead of the standard memory locations. We illustrate the results of the computation of the symbolic transfer functions in the following example:

Definition 9.4.4. Computation of symbolic transfer functions.

We consider the computation of the symbolic transfer functions in the case of Example 9.4.3. After abstraction of the registers, we get the expected results:

$$\begin{array}{rcl} \delta_{\iota_0^o, \iota^f} & = & \lfloor \mathbf{M}[\underline{i}] \leftarrow \mathbf{M}[\underline{i}] + 1 \rfloor \\ \delta_{\iota^f, \iota_7^o} & = & \lfloor \mathbf{M}[\underline{x}] \leftarrow \mathbf{M}[\underline{x}] + \mathbf{M}[\underline{t} + \mathbf{M}[\underline{i}]] \rfloor \end{array}$$

Obviously, these symbolic transfer functions between fictitious control states are very well fitted to the various certification algorithms stated in the next chapters.

9.4.4 Optimizations transforming paths

Many compilers carry out structure modifying optimizations such as loop unrolling and branch optimizations. These transformations reduce the time spent in branchings and interact well with the scheduling optimizations considered in Section 9.4.3). These transformations break the program point mapping $\Pi_{\mathbb{L}}$ in a different way: one source point may correspond to several assembly points (not to a sequence of points).

In this section, we focus on *loop unrolling*. This optimization consists in grouping two successive iterations of a loop, as is the case in the example below.

Definition 9.4.5. Loop unrolling.

We use the same syntax as for source programs for the sake of convenience and concision (the transformation envisaged here is similar to loop unrolling in assembly programs, up-to some details, which can be abstracted away in the same way as in the previous sections).

```
$\ell_0^n$ :  $i := 0$;
$\ell_1^n$ :  while($i < 2n$) {
$\ell_2^n$ :        $B$;
$\ell_3^n$ :        $i := i + 1$; }
$\ell_4^n$ :  ...
```

(a) Initial program P_n

```
$\ell_0^o$ :    $i := 0$;
$\ell_{1,e}^o$ :  while($i < 2n$){
$\ell_{2,e}^o$ :        $B$;
$\ell_{3,e}^o$ :        $i := i + 1$;
$\ell_{1,o}^o$ :
$\ell_{2,o}^o$ :        $B$;
$\ell_{3,o}^o$ :        $i := i + 1$; }
$\ell_4^o$ :    ...
```

(b) Optimized program P_o

Figure 9.7: Loop unrolling

We present two programs in Figure 9.7: the initial, non optimized *program P_n (Figure 9.7(a)) consists in a loop with a counter i; the* optimized *code P_o (Figure 9.7(b)), with the loop unrolled. Indeed, one iteration in the loop of P_o corresponds to* two *iterations in the loop of P_n. We use the index e (resp. o) is used for control states corresponding to the even (resp. odd) iteration numbers.*
The source control state ℓ_2^n corresponds to two control states in P', namely $\ell_{2,e}^o$ and $\ell_{2,o}^o$); and the same for ℓ_3^n. We also duplicated ℓ_1^n into $\ell_{1,e}^o$ and $\ell_{1,o}^o$ for the sake of the example.

Example 9.4.5 presents the main difficulty with loop unrolling: the points inside the loop are not in direct correspondence with the program points of the initial program. In fact, a point in the loop of P_n corresponds to *two* points in P_o, so that there is no way to define a bijective $\Pi_{\mathbb{L}}$ function.

The solution consists in using the trace partitioning framework of Chapter 4 so as to define a non-standard semantics for P_n with the following properties:

- the non-standard semantics should mimics the behavior of the transformed program;
- it should also be an abstraction of the standard semantics.

This amounts to stating the correctness of the compilation of some complete partition (or complete covering) of P_{s} (Definition 4.2.2) into P_{c}, following Definition 9.3.1. As a consequence, stating the correctness of a transformation like loop unrolling requires only a straightforward extension of our definition for compilation correctness.

Let us apply this extended definition to Example 9.4.5:

Definition 9.4.6. Compilation up-to partitioning.

We state the correctness of the transformation considered in Example 9.4.5.
In fact P_o is a complete partition of P_n, with the following notations:

- *the set of tokens T is $\{t, t_e, t_o\}$ (t is the "default" token, t_e corresponds to "even", and t_o to "odd");*

- *the control states of P_o are defined as follows:*

$$\begin{array}{rclcrcl}
\ell_0^o & = & (\ell_0^n, t) & & \ell_{1,o}^o & = & (\ell_1^n, t_o) \\
\ell_{1,e}^o & = & (\ell_1^n, t_e) & & \ell_{2,o}^o & = & (\ell_2^n, t_o) \\
\ell_{2,e}^o & = & (\ell_2^n, t_e) & & \ell_{3,o}^o & = & (\ell_3^n, t_o) \\
\ell_{3,e}^o & = & (\ell_3^n, t_e) & & \ell_4^o & = & (\ell_4^n, t)
\end{array}$$

- *the forget function τ maps any token to the "trivial" token t_ϵ (Section 4.2.1).*

As a consequence, the extended definition applies immediately.

The correctness of many other loop and path transformations could be proved correct in the same way.

9.4.5 Structure modifying optimizations

The list of optimizations which could be studied here could grow infinite.

For instance, we gave more general definitions of variable mappings in [Riv04b], so as to formalize structure modifying optimizations, which may change the flows of values, such as *loop reversal*. We do not claim that all optimizations could be formalized with the abstractions and mappings introduced in this chapter; however we believe that our methodology is general enough, so that a large number of optimizations can be dealt with.

In the next two chapters, we focus on compilation certification. Basically, the goal of the $\alpha_{\Pi\langle\mathbb{X}_s,\mathbb{L}_s\rangle}$ and $\alpha_{\Pi\langle\mathbb{X}_c,\mathbb{L}_c\rangle}$ abstractions is to establish what the compilation certification should care about, while the mappings $\Pi_{\mathbb{L}}, \Pi_{\mathbb{X}}$ establish the relation between both programs.

Chapter 10

Invariant Translation and Checking

We propose to compile invariants computed during an analysis of the source program. This approach should allow be more efficient and produce more precise invariants than the analysis of the compiled program. This technique can be considered a generalization of the Proof Carrying Code technique [Nec97]. We formalize it inside the framework for defining compilation introduced in Chapter 9 in Section 10.2.

Moreover, we discuss the issue of the independent checking of the translated invariant in Section 10.3, which should provide a higher level of confidence in the result.

Last, we provide implementation feed-back.

10.1 Principle and Related Work

The purpose of this chapter is to compute abstract invariants for compiled programs, so as to prove their safety. Moreover, the safety properties of interest may express more simply at the assembly level: in particular memory errors depend on the assembly memory model and the nature of the code generated, since the C norm leaves many behaviors "undefined". For instance, we wish to compute invariants akin to those produced by Astrée [BCC$^+$02, BCC$^+$03a, CCF$^+$05], and rely on them in order to check that critical operations such as memory accesses or arithmetic operations never cause any run-time error.

In theory, the static analysis presented in Section 3.1 extends to the assembly language presented in Section 9.2. However, this approach requires solving several practical issues:

- Compilation induces a **loss of control structure**. In particular, conditions, loops and conditional statements are compiled into graphs with gotos. Therefore, the computation of the least fixpoint inherent in the static analysis requires computing an adequate set of widening points [Bou93]. Moreover, a strategy for limiting the storage of local invariants would be required in order to allow the analysis to scale up (see [HDT87] for details). This would amount to recovering the control structure, which was lost at compile time.

- Compilation causes the **expansion of data-structures**: arrays, enumerations, structures, union types are all translated into series of bytes. Therefore, an assembly level analyzer would deal with low level data-structures only.
- Assembly level invariants are **tedious to read**. In case the analyzer does not conclude the code is safe, a diagnosis should be made for the alarms produced by the analysis. However, an assembly level analyzer would produce assembly level invariants, which would not be very helpful for the user. More generally, the invariants should be human readable and allow for a straightforward interpretation. As a consequence, we would need to relate the results of the analysis to the source program, so as to let the user understand whether the code indeed contains a bug. Again, this would amount to recovering the structure lost at compile time.

These arguments plead in favor of using the results of a source analysis for the certification of the compiled code. Of course, this would not be possible for any static analysis. For instance, analyses for determining low level properties of assembly programs cannot be done at the source level. For instance, cache prediction [AFMW96, FMW97], pipeline behavior [TF98] and worst case execution time [TFW00] analyses were implemented; they do not suffer the problems mentioned above and yield very good results (namely, precise bounds for worst case execution time). We can also cite the analysis of memory accesses in executables in [BR04], aimed at checking the security of assembly programs. The Java Virtual Machine [LY05] (aka JVM, developed by SUN) provides another common example of assembly level analysis. Indeed, the JVM performs a series of data-flow analyses in order to check the compliance of bytecode files with the Java bytecode standard, before running them. Among the properties verified, we can cite the type safety, the right definition of the stack (size and type of the arguments)...

However, at the time we write this thesis, we are aware of no analysis for determining high level properties at the assembly level, such as precise bounds on the range of variables.

We propose to translate the results produced by a source analyzer such as ASTRÉE into invariants for the compiled program. Such a translation is based on the relation between the source and the compiled program defined by $\Pi_{\mathbb{L}}, \Pi_{\mathbb{X}}$. Moreover, we perform an independent checking of safety of the translated invariants, justified by the soundness of fixpoint checking (Theorem 2.3.3). The goal of this independent checking is not to rely on the soundness of the compiler.

This solution has several advantages. First, it allows to carry out the fixpoint computation at the source level, using the most structured code: a fine iteration strategy is needed in order to infer precise invariants, which is easier to do at the source level (as in Section 3.2.5). Second, it allows to cope with alarms and to interpret the analysis results at the source level. Last, the final checking should give a sufficient level of confidence in the analysis results.

This approach presents some strong similarities with *Proof Carrying Code* systems (PCC), which were introduced in [Nec97], as a means to compile types with programs. The initial goal of PCC was to let a source producer provide some evidence of the safety of the code (i.e., the compliance with a pre-defined safety policy); the code consumer would run a program only after checking the safety using the annotations provided by the code producer. The implementation of a certifying compiler, producing types together with the compiled code is described in [NL98]. However, a significant difference is that PCC systems usually assume that the traduction of the type information is performed by the compiler, which we cannot do, since we use a generic compiler: by contrast, we assume the compilation correct in the sense of Definition 9.3.1 and base the translation procedure upon the mappings $\Pi_{\mathbb{L}}, \Pi_{\mathbb{X}}$. Finally, other authors extended the PCC framework. For instance, [App01] focused on the reduction of the trusted base, i.e. of the amount of code the soundness of the PCC system depends on. It is based on the reduction of the set of axioms to use for the type checking to a minimal number of rules.

Similarly, the Java bytecode compiler embeds enough information in the bytecode programs, so as to reduce the inference task that the java bytecode verifier should perform.

Other authors focused on the definition of Typed Intermediate Language, (TIL) such as [MTC$^+$96, TMC$^+$96], as a means to keep information about source ML programs in order to make further optimizations possible and trustable. Basically, well-typed programs should not produce some kinds of errors (the memory allocation should be safe). The principle of Typed Intermediate Languages is to require transformations (compilation, optimization) to preserve types, which entails that they preserve the safety. This methodology was extended to a Typed Assembly Language (TAL) in [MCG$^+$99]: The purpose of this work was also to design a safe compiler for a type-safe subset of C. This compiler is also supposed to translate types together with programs. Among the applications of these typed languages, we can cite the definition and trustable compilation of type-safe C-like languages, such as Cyclone [GMJ$^+$02] and CCured [NMW02]. The TAL technique was extended by [XH01] so as to rely on more expressive types, i.e. dependent types, in order to verify more complex properties.

The implementation of invariant translation and invariant checking in the abstract interpretation framework was presented in [Riv03, Riv04a], as a means to design an assembly code analyzer similar to ASTRÉE [BCC$^+$03a], which would work for compiled code. This chapter follows the presentation of these papers.

Section 10.2 describes and proves correct the invariant translation procedure; Section 10.3 discusses the main issues of the invariant checking.

10.2 The Invariant Translation

10.2.1 Invariant translation for the reduced compiled program

Assumptions: In this section, we consider a source program P_{s} and a compiled program P_{c}. Moreover, we assume that the compilation of P_{s} into P_{c} is sound in the sense of Definition 9.3.3. In particular, the correctness of the compilation guarantees the existence of two mappings:

- $\Pi_{\mathbb{L}} : \overline{\mathbb{L}}_{\mathrm{s}} \rightarrow \overline{\mathbb{L}}_{\mathrm{c}}$ where $\overline{\mathbb{L}}_{\mathrm{s}} \subseteq \mathbb{L}_{\mathrm{s}}$ and $\overline{\mathbb{L}}_{\mathrm{c}} \subseteq \mathbb{L}_{\mathrm{c}}$;
- $\Pi_{\mathbb{X}} : \overline{\mathbb{X}}_{\mathrm{s}} \rightarrow \overline{\mathbb{X}}_{\mathrm{c}}$ where $\overline{\mathbb{X}}_{\mathrm{s}} \subseteq \mathbb{X}_{\mathrm{s}}$ and $\overline{\mathbb{X}}_{\mathrm{c}} \subseteq \mathbb{X}_{\mathrm{c}}$.

As usual, we let $P_{\mathrm{s}}^{\mathrm{r}}$ (resp. $P_{\mathrm{c}}^{\mathrm{r}}$) denote the reduced program associated to P_{s} (resp. P_{c}). Moreover, the soundness of compilation guarantees that $\Pi_\Sigma(\llbracket P_{\mathrm{s}}^{\mathrm{r}} \rrbracket) = \llbracket P_{\mathrm{c}}^{\mathrm{r}} \rrbracket$, where Π_Σ is the trace mapping derived from $\Pi_{\mathbb{L}}$ and $\Pi_{\mathbb{X}}$, thanks to the statement based on reduced programs (Definition 9.3.3).

We also perform a static analysis of the source program. More precisely, we write $D^\sharp_{\mathbb{M},\mathrm{s}}$ for the abstract domain for representing source stores, and $\gamma^\sharp_{\mathbb{M},\mathrm{s}} : D^\sharp_{\mathbb{M},\mathrm{s}} \rightarrow \mathcal{P}(\mathbb{M}_{\mathrm{s}})$ the corresponding concretization function. As usual, we let $D^\sharp_{\mathrm{s}}$ denote the domain for abstracting sets of traces, defined as in Section 3.1.1 by $D^\sharp_{\mathrm{s}} = \mathbb{L}_{\mathrm{s}} \rightarrow D^\sharp_{\mathbb{M},\mathrm{s}}$ and γ_{s} denote the concretization function $\gamma_{\mathrm{s}} : D^\sharp_{\mathrm{s}} \rightarrow \mathcal{P}(\Sigma_{\mathrm{s}})$. We write $\mathfrak{I}_{\mathrm{s}} \in D^\sharp_{\mathrm{s}}$ for the invariant produced by the static analysis and remember the soundness condition:

$$\forall \langle \ldots, (l, \rho) \rangle \in \llbracket P_{\mathrm{s}} \rrbracket,\ \rho \in \gamma^\sharp_{\mathbb{M},\mathrm{s}}(\mathfrak{I}_{\mathrm{s}}(l))$$

We consider in this section a simplified version of the example of Figure 9.3 (note that the initial value of i is 0 instead of -1):

Definition 10.2.1. Compiled program.

We let P_{s} and P_{c} be the programs displayed respectively in Figure 10.1(a) and in Figure 10.1(b). This compilation is correct in the sense of Section 9.3.3, with:

- *for the source program, $\overline{\mathbb{X}}_{\mathrm{s}} = \mathbb{X}_{\mathrm{s}} = \{i\}$, and $\overline{\mathbb{L}}_{\mathrm{s}} = \mathbb{L}_{\mathrm{s}} = \{l_0^s, l_1^s, l_2^s, l_3^s, l_4^s\}$;*
- *for the assembly program, $\overline{\mathbb{X}}_{\mathrm{c}} = \{\mathbf{M}[\underline{i}]\}$, and $\overline{\mathbb{L}}_{\mathrm{c}} = \{l_0^c, l_2^c, l_6^c, l_{10}^c, l_{11}^c\}$;*
- *the control state mapping:*

$$\Pi_{\mathbb{X}} : \begin{array}{ccc} l_0^s & \mapsto & l_0^c \\ l_1^s & \mapsto & l_2^c \\ l_2^s & \mapsto & l_6^c \\ l_3^s & \mapsto & l_{10}^c \\ l_4^s & \mapsto & l_{11}^c \end{array}$$

- *the memory location mapping: $\Pi_{\mathbb{X}} : i \mapsto \mathbf{M}[\underline{i}]$*

In the following, we consider a simple interval analysis: $D^\sharp_{\mathbb{M},\mathrm{s}} = \mathbb{X} \rightarrow \mathsf{Intervals}\langle\mathbb{I}\rangle$, where $\mathsf{Intervals}\langle\mathbb{I}\rangle$ collects all the intervals of values ranging in the set of machine integers $\mathbb{I}$ and:

$$\gamma_{\mathsf{Intervals}\langle\mathbb{I}\rangle} : \begin{array}{lcl} \mathsf{Intervals}\langle\mathbb{I}\rangle & \rightarrow & \mathcal{P}(\mathbb{M}) \\ \Phi & \rightarrow & \{\rho \in \mathbb{M} \mid \forall x \in \mathbb{X},\ \rho(x) \in \Phi(x)\} \end{array}$$

We choose intervals for the sake of simplicity; however, the results in this section would obviously generalize to other domains.

The most basic interval analyzer would compute the following invariants:

Control state	*Interval for i*
l_0^s	$\mathbb{I}$
l_1^s	$[0, 100]$
l_2^s	$[0, 99]$
l_3^s	$[1, 100]$
l_4^s	$[100, 100]$

```
l_0^s :  int i := 0;
l_1^s :  while(i < 100) {
l_2^s :      i := i + 1;
l_3^s :  }
l_4^s :  ...
```

(a) Source program P_{s}

```
l_0^c  :  li      gpr_0, 0
l_1^c  :  store   gpr_0, i(0)
l_2^c  :  load    gpr_0, i(0)
l_3^c  :  li      gpr_1, 100
l_4^c  :  cmp     cr_0, gpr_0, gpr_1
l_5^c  :  bc(≥)   cr_0, l_11^c
l_6^c  :  load    gpr_0, x(0)
l_7^c  :  li      gpr_1, 1
l_8^c  :  add     gpr_2, gpr_0, gpr_1
l_9^c  :  store   gpr_2, x(0)
l_10^c :  b       l_2^c
l_11^c :  ...
```

(b) Assembly program P_{c}

In the following, we attempt to derive local invariants for P_{c} from $\mathfrak{I}_{\mathrm{s}}$; we consider the case of control states in the reduced program first.

Properties of the reduced compiled program: Let $l_c \in \overline{\mathbb{L}}_{\mathrm{c}}$, $\rho_c \in \overline{\mathbb{M}}_{\mathrm{c}}$, and a trace $\sigma_c = \langle \ldots, (l_c, \rho_c)\rangle \in [\![P_{\mathrm{c}}^{\mathrm{r}}]\!]$. The correctness of the compilation guarantees the existence of a trace $\sigma_s \in [\![P_{\mathrm{s}}^{\mathrm{r}}]\!]$, such that $\Pi_\Sigma(\sigma_s) = \sigma_c$. As a consequence, there exist $l_s \in \overline{\mathbb{L}}_{\mathrm{s}}$, $\rho_s \in \overline{\mathbb{M}}_{\mathrm{s}}$, such that $\sigma_s = \langle \ldots, (l_s, \rho_s)\rangle$. Hence, $\Pi_{\mathbb{L}}(l_s) = l_c$, and $\rho_s = \rho_c \circ \Pi_{\mathbb{X}}$.

Moreover, the soundness of the analysis entails that $\rho_s \in \gamma^\sharp_{\mathbb{M},\mathrm{s}}(\mathfrak{I}_{\mathrm{s}}(l_s))$, i.e. $\rho_c \circ \Pi_{\mathbb{X}} \in \gamma^\sharp_{\mathbb{M},\mathrm{s}}(\mathfrak{I}_{\mathrm{s}}(l_s))$.

The function $\Pi_{\mathbb{X}}$ is a bijection, therefore we can compute its inverse $(\Pi_{\mathbb{X}})^{-1}$. As a consequence $\rho_c \in (\Pi_{\mathbb{X}})^{-1} \circ \gamma^\sharp_{\mathbb{M},\mathrm{s}}(\mathfrak{I}_{\mathrm{s}}(l_s))$.

Therefore, we can derive an invariant for the restricted compiled program:

Theorem 10.2.1. Invariant for $P_{\mathrm{c}}^{\mathrm{r}}$.

Let $\mathfrak{I}_{\mathrm{c}}^{\mathrm{r}}$ be the invariant defined by:

$$\begin{array}{rccl} \mathfrak{I}_{\mathrm{c}}^{\mathrm{r}} : & \overline{\mathbb{L}}_{\mathrm{c}} & \rightarrow & D^\sharp_{\mathbb{M},\mathrm{s}} \\ & l_c & \mapsto & \mathfrak{I}_{\mathrm{s}}((\Pi_{\mathbb{L}})^{-1}(l_c)) \end{array}$$

Then, $\mathfrak{I}_{\mathrm{c}}^{\mathrm{r}}$ is a sound invariant for $P_{\mathrm{c}}^{\mathrm{r}}$:

$$\forall \langle \ldots, (l_c, \rho_c)\rangle \in [\![P_{\mathrm{c}}^{\mathrm{r}}]\!],\ \rho_c \in (\Pi_{\mathbb{X}})^{-1} \circ \gamma^\sharp_{\mathbb{M},\mathrm{s}}(\mathfrak{I}_{\mathrm{c}}^{\mathrm{r}}((\Pi_{\mathbb{L}})^{-1}(l_c)))$$

Proof.

Follows from the above remark, about ρ_c. $\Box$

Theorem 10.2.1 provides the skeleton of an invariant for the assembly program; however, it fails to deliver any precise information about the values of any variable at any point in $\mathbb{L}_c \setminus \overline{\mathbb{L}}_c$. Refining the result of Theorem 10.2.1 is the purpose of the following subsection.

Definition 10.2.2. Example 10.2.1 continued.

As a consequence of theorem 10.2.1, we deduce the following invariant for P_c^r:

Control state	*Interval for* $\mathbf{M}[\underline{i}]$
l_0^c	$\mathbb{I}$
l_1^c	$[0, 100]$
l_2^c	$[0, 99]$
l_3^c	$[1, 100]$
l_4^c	$[100, 100]$

10.2.2 Invariant translation for the whole compiled program

Let $D^\sharp_{\mathbb{M},c}$ be a domain for representing sets of stores for the target language, and $\gamma^\sharp_{\mathbb{M},c} : D^\sharp_{\mathbb{M},c} \rightarrow \mathcal{P}(\mathbb{M}_c)$ be the associated concretization function. In practice, the domain $D^\sharp_{\mathbb{M},c}$ is similar to $D^\sharp_{\mathbb{M},s}$: for instance, in case the source analysis generates interval invariants, the translated invariants also consists in interval constraints. Moreover, we assume that we are able to compute abstract transfer functions for the target language: we assume that, for all $l, l' \in \mathbb{L}_c$, $d \in D^\sharp_{\mathbb{M},c}$, $\rho \in \gamma^\sharp_{\mathbb{M},c}(d)$, $\rho' \in \mathbb{M}_c$ such that $(l, \rho) \rightarrow (l', \rho')$, then $\rho' \in \delta^\sharp_{l,l'}(d)$.

We propose to derive from $\mathfrak{I}_c^r$ an invariant for P_c in two steps: first, we envisage the case of a control state in $\overline{\mathbb{L}}_c$; second, we consider a control state in $\mathbb{L}_c \setminus \overline{\mathbb{L}}_c$.

Case of a point l_c in $\overline{\mathbb{L}}_c$: Then, $\mathfrak{I}_c^r(l_c)$ provides us with invariants for variables in $\overline{\mathbb{X}}_c$, but no information about the other memory locations in the compiled program: for instance, it does not tell us anything about the registers. Therefore, we map $\mathfrak{I}_c^r(l_c)$ into an invariant $\mathfrak{I}_c(l_c)$, which is defined in a domain expressing constraints for variables in $\mathbb{X}_c$, using a kind of "injection function" inject. In all cases we are aware of, this step is straightforward, since abstract values denote collections of constraints, and $\mathfrak{I}_c(l_c)$ simply stands for the same set of constraints as $\mathfrak{I}_c^r(l_c)$, but in a richer domain (since more variables are allowed).

Definition 10.2.3. Example 10.2.2 continued.

The abstract domain $D^\sharp_{\mathbb{M},c}$ maps condition registers into subsets of $\mathbb{C}$ (non relational approximation) and general purpose registers and memory locations into integer intervals.

The case of the condition registers is not so obvious: we may naively consider that a set of possible condition values would be adequate.
In this case, at any point in $\overline{\mathbb{L}}_{\mathrm{c}}$, the invariant $\mathfrak{I}_{\mathrm{c}}$ should store:

- *intervals similar to those in Example 10.2.2 for* $\mathbf{M}[\underline{i}]$*;*
- *no information for general purpose registers, i.e., the interval* $\mathbb{I}$*;*
- *no information for the condition registers, i.e., the abstract value* $\mathbb{C}$*.*

As a consequence, we get at this stage the following invariant:

control state	$\mathbf{M}[\underline{i}]$	$\mathbf{gpr}_0$	$\mathbf{gpr}_1$	$\mathbf{cr}_0$
l_0^c	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{C}$
l_2^c	$[0, 100]$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{C}$
l_6^c	$[0, 99]$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{C}$
l_{10}^c	$[1, 100]$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{C}$
l_{11}^c	$[100, 100]$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{C}$

Case of a point in $\mathbb{L}_{\mathrm{c}} \setminus \overline{\mathbb{L}}_{\mathrm{c}}$: Let us consider the case of a point $l_c \notin \overline{\mathbb{L}}_{\mathrm{c}}$ now.

There exists at most a finite number of feasible paths $p = l_0^c \cdot \ldots \cdot l_n^c$ such that $l_0^c \in \overline{\mathbb{L}}_{\mathrm{c}}$, $\forall i > 0,\ l_i^c \notin \overline{\mathbb{L}}_{\mathrm{c}}$ and $l_c = l_n^c$ (a path is feasible if and only if there exists a real program execution following it). Let $\mathcal{P}$ be the set containing all such paths.

Let us consider a trace σ_c in $[\![P_{\mathrm{c}}]\!]$ ending in l_c: there exists $\rho_c \in \mathbb{M}_{\mathrm{c}}$, such that $\sigma_c = \langle \ldots, (l_c, \rho_c)\rangle$. This trace starts at the entry point in the program so it encounters at least one control state in $\overline{\mathbb{L}}_{\mathrm{c}}$. Therefore, we consider the last such control state in σ_c and let l_0^c denote it. Furthermore, σ_c follows a path in $\mathcal{P}$ after that point. Let us write $l_0^c \cdot \ldots \cdot l_n^c$ for that path (where $l_0^c \in \overline{\mathbb{L}}_{\mathrm{c}}$, $l_n^c = l_c$); then, $\sigma_c = \langle \ldots (l_0^c, \rho_0^c), \ldots, (l_n^c, \rho_n^c)\rangle$. The soundness of the translated invariant for $l_0^c \in \overline{\mathbb{L}}_{\mathrm{c}}$ entails that $\rho_0^c \in \gamma^\sharp_{\mathbb{M},\mathrm{c}}(\mathfrak{I}_{\mathrm{c}}(l_0^c))$. The soundness of the transfer functions $\delta^\sharp_{\cdot,\cdot}$ implies that:

$$\rho_c \in \delta^\sharp_{l_{n-1}^c, l_n^c} \circ \ldots \delta^\sharp_{l_0^c, l_1^c}(\mathfrak{I}_{\mathrm{c}}(l_0^c))$$

Therefore $\rho_c \in \mathfrak{I}_{\mathrm{c}}(l_n^c)$, where:

$$\mathfrak{I}_{\mathrm{c}}(l_n^c) = \bigsqcup \left\{ \delta^\sharp_{l_{n-1}^c, l_n^c} \circ \ldots \circ \delta^\sharp_{l_0^c, l_1^c}(\mathfrak{I}_{\mathrm{c}}(l_0^c)) \mid l_0^c \cdot \ldots \cdot l_n^c \in \mathcal{P} \right\}$$

(where $\mathfrak{I}_{\mathrm{c}}(l_0^c)$ is defined as above since $l_0^c \in \overline{\mathbb{L}}_{\mathrm{c}}$)

To summarize:

Definition 10.2.1. Translated invariant.

We let the translated invariant be defined by:

- *if $l^c \in \overline{\mathbb{L}}_{\mathrm{c}}$, then:*

$$\mathfrak{I}_{\mathrm{c}}(l^c) = \mathit{inject}(\mathfrak{I}_{\mathrm{c}}^{\mathrm{r}}(l^c))$$

- *if* $\ell^c \notin \overline{\mathbb{L}}_{\mathrm{c}}$*, we define* $\mathcal{P}$ *as above and:*

$$\mathfrak{I}_{\mathrm{c}}(\ell^c) = \bigsqcup \left\{ \delta^\sharp_{\ell^c_{n-1},\ell^c_n} \circ \ldots \circ \delta^\sharp_{\ell^c_0,\ell^c_1}(\mathit{inject}(\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}(\ell^c_0))) \mid \ell^c_0 \cdot \ldots \cdot \ell^c_n \in \mathcal{P} \wedge \ell^c_n = \ell^c \right\}$$

Theorem 10.2.2. Soundness of the translated invariant.

First, we sum up the assumptions made in this section:

- *the invariant* $\mathfrak{I}_{\mathrm{s}}$ *soundly approximates* $[\![P_{\mathrm{s}}]\!]$*;*
- *the compilation of* P_{s} *into* P_{c} *is sound.*

Then, the translated invariant $\mathfrak{I}_{\mathrm{c}}$ *(Definition 10.2.1) is sound:*

$$\forall \langle \ldots, (\ell, \rho) \rangle \in [\![P_{\mathrm{c}}]\!],\ \rho \in \gamma^\sharp_{\mathbb{M},\mathrm{c}}(\mathfrak{I}_{\mathrm{c}}(\ell))$$

Proof.

The soundness follows from the two previous paragraphs. □

Theorem 10.2.2 provides a sound way of deriving an invariant $\mathfrak{I}_{\mathrm{c}}$ for the compiled program from an invariant for the source program. We illustrate the result in our example:

Definition 10.2.4. Example 10.2.3 continued.

The table below displays the translated invariant $\mathfrak{I}_{\mathrm{c}}$*:*

control state	$\mathbf{M}[\underline{i}]$	$\mathbf{gpr}_0$	$\mathbf{gpr}_1$	$\mathbf{cr}_0$
ℓ^c_0	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{C}$
ℓ^c_1	$\mathbb{I}$	$[0,0]$	$\mathbb{I}$	$\mathbb{C}$
ℓ^c_2	$[0,100]$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{C}$
ℓ^c_3	$[0,100]$	$[0,100]$	$\mathbb{I}$	$\mathbb{C}$
ℓ^c_4	$[0,100]$	$[0,100]$	$[100,100]$	$\mathbb{C}$
ℓ^c_5	$[0,100]$	$[0,100]$	$[100,100]$	$\{\mathrm{LT},\mathrm{EQ}\}$
ℓ^c_6	$[0,99]$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{C}$
ℓ^c_7	$[0,99]$	$[0,99]$	$\mathbb{I}$	$\mathbb{C}$
ℓ^c_8	$[0,99]$	$[0,99]$	$[1,1]$	$\mathbb{C}$
ℓ^c_9	$[0,99]$	$[1,100]$	$[1,1]$	$\mathbb{C}$
ℓ^c_{10}	$[1,100]$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{C}$
ℓ^c_{11}	$[100,100]$	$\mathbb{I}$	$\mathbb{I}$	$\mathbb{C}$

In many regards, this invariant is not optimal. For instance, no information about the value of $\mathbf{cr}_0$ *is inferred for* $\ell^c_6, \ldots, \ell^c_{11}$*, even though we might expect to find some. This is due to the fact we do not perform a global analysis of the target program.*

A major drawback of Theorem 10.2.2 is that the proof *assumes* the soundness of the compilation, despite one of the main reasons for analyzing the assembly code instead of the source code was to certify the compiled code even if it is produced by a non-trusted compiler. As a consequence, we will consider the problem of *checking* the translated invariant in an independent way: this will be the purpose of Section 10.3.

10.2.3 Translated invariant and program reduction

As we pointed out in Section 9.4, other forms of program reduction may be required:

- the source program may be reduced as well (e.g., in order to cope with the removal of dead variables): in this case, we need to forget all constraints about the variables in $\overline{\mathbb{X}} \setminus \overline{\mathbb{X}}_{\mathrm{s}}$, by applying a forget operator forget, which we introduced in Section 3.1.1;
- more complicated abstractions might be involved in the definition of correctness of compilation, in the case of more complex optimizations: then, the corresponding abstractions should be applied in the abstract level.

We followed this methodology in order to extend the invariant translation algorithm to various optimizations in [Riv04b]. In particular, the nature of the mapping between source and compiled programs conditions the nature of the invariants, which can be translated: for instance, optimizations such as scheduling may impede the translation of relational invariants if two variables cannot be made available at a same point.

Moreover, some optimizations allow for the traduction of *finer* invariants. This is the case of loop unrolling (Section 9.4.4): a partitioning analysis distinguishing even and odd iterations (Chapter 4) would produce more precise invariants, which can be translated exactly.

10.3 Invariant Checking

We proved the soundness of the translated invariant in Theorem 10.2.2, under the assumption that the compiler is sound. Obviously, we do not wish to rely on this assumption, since the compiler may be wrong. Therefore, we consider the *independent* checking of the translated invariant now: in this section, we no longer assume the compiler be correct, or even the source invariant be sound.

10.3.1 Principle of invariant checking

We propose to follow the fixpoint checking method presented in Theorem 2.3.3: if F is a monotone concrete semantic function, $F^\sharp$ is a sound abstract semantic function, approximating F with respect to a monotone concretization function γ, and d is an abstract element such that $F^\sharp(d) \sqsubseteq d$, then $\mathbf{lfp}F \subseteq \gamma(d)$. The invariant checking theorem below corresponds to a slight improvement upon Theorem 2.3.3.

We could either perform the checking of $\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}$ or of $\mathfrak{I}_{\mathrm{c}}$. In the following, we perform the verification of $\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}$: this approach makes sense, since $\mathfrak{I}_{\mathrm{c}}$ is computed from $\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}$.

Theorem 10.3.1. Invariant checking.

Let $\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}} \in (\overline{\mathbb{L}}_{\mathrm{c}} \to D^{\sharp}_{\mathbb{M},\mathrm{s}})$ *be a* candidate invariant *for the compiled, reduced program. In case the property below holds, then the invariant* $\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}$ *is a sound approximation of* $[\![P^{\mathrm{r}}_{\mathrm{c}}]\!]$*:*

$$\left.\begin{array}{l} \textit{for all feasible path } l_0 \cdot \ldots \cdot l_n, \\ l_0 \in \overline{\mathbb{L}}_{\mathrm{c}} \\ l_n \in \overline{\mathbb{L}}_{\mathrm{c}} \\ \forall i \in (\!|1, n-1|\!),\ l_i \notin \overline{\mathbb{L}}_{\mathrm{c}} \end{array}\right\} \Longrightarrow \delta^{\sharp}_{l_{n-1},l_n} \circ \ldots \circ \delta^{\sharp}_{l_0,l_1}(\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}(l_0)) \sqsubseteq \mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}(l_n) \qquad (10.1)$$

Proof.

Applying the result of Theorem 2.3.3 would require a slightly different (and more approximate) definition for $F^{\sharp}$:

$$F^{\sharp}: \quad \mathfrak{I} \quad \mapsto \quad \lambda(l_n \in \overline{\mathbb{L}}_{\mathrm{c}}) \cdot \bigsqcup\{\delta^{\sharp}_{l_{n-1},l_n} \circ \ldots \circ \delta^{\sharp}_{l_0,l_1}(\mathfrak{I}(l_0)) \mid l_0 \in \overline{\mathbb{L}}_{\mathrm{c}} \wedge \forall i \in (\!|1, n-1|\!),\ l_i \notin \overline{\mathbb{L}}_{\mathrm{c}}\}$$

Instead, we avoid to compute the abstract join (which is a major source of imprecision); therefore, we cannot deduce Theorem 10.3.1 directly from Theorem 2.3.3, even though the principle of the proof is similar.

In the following, we assume that the checking mentioned in Theorem 10.3.1 succeeds.

Let $l \in \overline{\mathbb{L}}_{\mathrm{c}}$, and $\rho \in \overline{\mathbb{M}}_{\mathrm{c}}$. We assume that $l \in \gamma^{\sharp}_{\mathbb{M},\mathrm{c}}(\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}(l))$. We propose to show that any transition in the restricted compiled program from (l, ρ) leads to another state, which is safely approximated by $\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}$.

Let $(l', \rho') \in \overline{\mathbb{S}}_{\mathrm{c}}$, such that there is a transition $(l, \rho) \to (l', \rho')$ in the restricted, compiled program. Theorem 9.3.1 implies that there exists a trace $\sigma_c = \langle (l, \rho_c), \ldots, (l', \rho'_c) \rangle$ of the compiled program, such that $\Pi^{\mathrm{trace}}_{\overline{\mathbb{X}}_{\mathrm{c}},\overline{\mathbb{L}}_{\mathrm{c}}}(\sigma_c) = \langle (l, \rho), (l', \rho') \rangle$. We let $p = l \cdot l_0 \cdot \ldots \cdot l_n \cdot l'$ be the path underlying σ_c. The soundness of the local transfer functions ensures that $\rho' \in \gamma^{\sharp}_{\mathbb{M},\mathrm{c}}(\delta^{\sharp}_{l_n,l'} \circ \ldots \circ \delta^{\sharp}_{l,l_0}(\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}(l)))$. Moreover, the success of the invariant checking insures that $\delta^{\sharp}_{l_n,l'} \circ \ldots \circ \delta^{\sharp}_{l,l_0}(\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}(l)) \sqsubseteq \mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}(l')$; the monotonicity of $\gamma^{\sharp}_{\mathbb{M},\mathrm{c}}$ implies that $\rho' \in \gamma^{\sharp}_{\mathbb{M},\mathrm{c}}(\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}(l'))$.

Similarly, we can prove that the initial states for the restricted program are in the concretization of $\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}$.

As a conclusion, it follows that $\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}$ over-approximates the semantics of the restricted, compiled program. □

A major drawback of the checking is that it requires implementing almost a full abstract interpreter for the target language: the main part which does not need to be implemented is the abstract post-fixpoint engine (i.e., we do not have to implement the iterator, to choose a widening strategy...). However, the most important issues with the invariant checking procedure are described in Section 10.3.2.

10.3.2 Issues with the precision of transfer functions

Incompleteness of the abstract transfer functions: Theorem 10.3.1 provides a sound invariant checking procedure; however, we do not prove the completeness of the procedure. In fact, the invariant checking procedure described by Theorem 10.3.1 is *not complete*. Similarly, if γ is monotone, and $F \circ \gamma \subseteq \gamma \circ F^\sharp$ but the verification condition $F^\sharp(x) \sqsubseteq x$ fails (for instance, if $F^\sharp(x)$ and x are not comparable, or due to a lack of local monotonicity of $F^\sharp$), then it does *not* entail that $\gamma(x)$ is not a valid over-approximation for the concrete least fixpoint.

Definition 10.3.1. Incompleteness of the invariant checking.
Let us assume that concrete and abstract values are natural integers, $F : \mathbb{N} \rightarrow \mathbb{N}, x \mapsto 4$, $F^\sharp : \mathbb{N} \rightarrow \mathbb{N}, x \mapsto x + 4$ *and* $\gamma : n \mapsto n$. *Then,* $F^\sharp(4) \not\leq 4$ *(so that the checking fails) even though* $F(4) = 4$ *(i.e., the "invariant"* 4 *is sound).*

Among the reasons, which may lead to the invariant checking to fail despite $\mathfrak{I}_{\mathrm{c}}^{\mathrm{r}}$ is sound, we can cite:

- the possible non-monotonicity of the transfer functions involved in the invariant checking;
- the imprecision of the transfer functions and of the abstract domain used for the analysis of the restricted, compiled program.

In the following of this subsection, we describe examples for precision issues, which may require the invariant checking to fail. These problems occur in practice, and led to the development of refined domains for the invariant checking to succeed in [Riv03, Riv04a].

Definition 10.3.2. Failure of invariant checking.
In particular, the checking of the translated invariant given in Example 10.2.2 fails at the entrance in the loop body. Indeed, the certification condition states that $\delta^\sharp_{l_5^c,l_6^c} \circ \ldots \circ \delta^\sharp_{l_2^c,l_3^c}(\mathfrak{I}_{\mathrm{c}}^{\mathrm{r}}(l_2^c)) \sqsubseteq \mathfrak{I}_{\mathrm{c}}^{\mathrm{r}}(l_6^c)$.
However, this condition amounts to $\delta^\sharp_{l_5^c,l_6^c}(\mathfrak{I}_{\mathrm{c}}(l_5^c)) \sqsubseteq \mathfrak{I}_{\mathrm{c}}^{\mathrm{r}}(l_6^c)$, *where* $\mathfrak{I}_{\mathrm{c}}$ *is given in Example 10.2.4 (up-to the abstraction of registers), and the latter condition fails, since the range for* $\mathbf{M}[\underline{i}]$ *is* $[0, 100]$ *at point* l_5^c*; it is* $[0, 99]$ *at point* l_6^c *(we note that the failure is observed for* $\mathbf{M}[\underline{i}]$ *which belongs to* $\overline{\mathbb{X}}_{\mathrm{c}}$*, so that we cannot blame the fact that we considered* $\mathfrak{I}_{\mathrm{c}}$ *for this failure).*
Indeed, the value of $\mathbf{M}[\underline{i}]$ *is not modified in this sequence of instruction, yet on the values in the range* $[0, 99]$ *go through the branch to* l_6^c *due to the test. The reason why the checking analysis does not remark this is the result of the lack of* relation *between the value of* $\mathbf{cr}_0$. *In fact, the condition is tested on a* copy *of* $\mathbf{M}[\underline{i}]$*, which is also impedes the invariant checking.*

Conditions: As we remarked in Example 10.3.2, the checking of conditions requires relations between the values of the condition registers and the other abstract values to be maintained. This issue was solved in [Riv04a] by a partitioning domain, where each possible value of the condition register is mapped into a value of $D^\sharp_{\mathbb{M},\mathsf{c}}$.

Use of copies: Most assembly operations affect *registers*. As a consequence, the evaluation of an assignment or of a condition requires the content of memory locations to be copied into registers. This copy might impede the invariant checking, as in the case of conditions (Example 10.3.2).

In particular, if the source analysis relies on the fact that some equality relations $x = y$ holds, then the assembly analysis should establish a relation of the form $\mathbf{gpr}_i = \mathbf{gpr}_j$ (where x and y are respectively copied into $\mathbf{gpr}_i$ and $\mathbf{gpr}_j$).

Definition 10.3.3. Symbolic simplification and invariant checking.

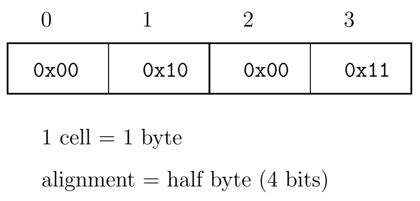
We pointed out that the simplification of symbolic transfer functions along paths could be used as a means to improve the precision of static analysis. In particular, we found that they solve the issue reported in Example 10.3.2.
Indeed, $\delta^\sharp_{\iota_5^c,\iota_6^c} \circ \ldots \circ \delta^\sharp_{\iota_2^c,\iota_3^c}$ can be simplified straightforwardly into (we abstract the registers away here):

$$\lfloor \mathbf{M}[\underline{i}] < 100 \; ? \; \iota \mid \Box \rfloor$$

This symbolic transfer function allows for a successful local invariant checking.
The reason why symbolic composition and simplification helps here is that it reconstructs the structure of the computations as in the source program and reduces the invariant checking to similar conditions as those used in the source analysis.

Low level operations: Other low level operations may require special care, including:

- the verification of **memory operations** requires the alignment of the addresses to be checked carefully. For instance, 32-bits architectures often use addresses corresponding to bytes: a cell of an integer array is 4 bytes long; therefore, the indexes should be congruent to 0 modulo 4. As a consequence, the reading of an integer array cell determined by an index in a range $[a, b]$ can be checked precisely only if the checker is able to prove that the index is congruent to 0 modulo 4; otherwise, the checker should also take into account the possibility of reading parts of two consecutive cells, which may return a very different result. We exemplify this situation in Figure 10.1.
- the low-level implementation of **data conversions** may involve complex properties. For instance, the implementation of the conversion of an integer value into a floating point value in the Power-PC architecture, is commonly compiled into a sequence of bitwise operations, subtractions and rounding (as show in Example 11.3.1). Precisely analyzing the whole conversion would require the sequence of operations to be recognized by the verifier as a conversion, since the abstract primitives for bitwise operations and subtractions would be very different from a conversion.



(c) Memory layout

Source: access to cell i, where $i \in [0, 1]$
Assembly: access to cell i, where $i \in [0, 2] \wedge i \equiv 0 \mod (2)$

analysis:	values read
no congruence analysis	$\{$0x0010, 0x1000, 0x0011, ...$\}$
alignments not handled precisely	value 0x1000mistakenly read
congruence analysis	$\{$0x0010, 0x0011$\}$
alignments handled precisely	no imprecision

(d) Read from the memory

Figure 10.1: Memory alignments and invariant checking

10.3.3 Practical experience

Implementation of a prototype: We implemented the invariant translation and invariant checking technique in a prototype in 2002 [Riv03, Riv04a].

- The source programs are written in C. The prototype was designed so as to handle families of synchronous applications (Section 5.1.1), in the same way as Astrée.
- The compiled programs are Power-PC assembly programs, including Stabs debugging information. During our experiment, we used the **gcc** compiler, even though any other compiler for the same architecture, with the same kind of debugging information could have been used instead.
  We restricted to non-optimized code.
- The purpose of the invariant translation was to prove the safety of the compiled programs, i.e., the absence of runtime errors (e.g., division by 0, wrong access to the memory) and of "user-defined wrong behaviors" (such as integer or floating-point overflows).

The structure of the C analyzer is rather similar to the early version of the Astrée analyzer [BCC$^+$02]. It uses a domain collecting interval constraints, clock constraints [BCC$^+$02], and trace partitioning (with part of the features of the partitioning domain introduced in Chapter 5). The invariant translator generates similar invariants.

The invariant checking required a few refinements to be implemented: partitioning with the value of the condition register, product with a domain for representing equality relations, product with a congruence domain [Gra89].

Benchmark and conclusion: The whole prototype was successfully ran on a few simple applications, including the 400 lines program mentioned in Section 5.3.3. In particular, *invariant checking was successful*, i.e., the translated invariant is proved correct independently from any assumption on the compiler and on the source code analysis (using the technique introduced in Theorem 10.3.1). Moreover, the translated invariant resulted in *only one false alarm*, which was present in the source analysis. This alarm was fixed later by a refinement of the source analyzer, so that we would expect an improved checker to produce no false alarm as well (the checker was not maintained by then, as explained below).

However, the amount of resources required by the tool were rather disappointing. We sum up the time and memory required for each step of the process in the table below. These measurements were done on an Intel Pentium III laptop (1 GHz), with 384 Mbytes of RAM.

Step	Time (s)	Memory (MB)
Source code analysis	2.5	15
Assembly code parsing and mapping construction	1.5	-
Invariant translation	4.5	20
Invariant checking	5.5	27

The main issue with the invariant checking is due to the memory requirement of the procedure. Indeed, large part of the data sharing ensured during the source analysis is lost in the translation, which causes the structures representing assembly invariants to use more memory and, to a lesser extent, the computation of the abstract transfer functions for the invariant checking to be slower.

By contrast, the very heavy use of data-sharing plays a considerable role in the source analysis; in particular, it allows for a lower memory usage, and for very fast operations, such as the computation of abstract joins.

A much more efficient translation and checking procedure could have been designed at the cost of the ease of maintenance of the invariant checker. However, the abstract domains developed in ASTRÉE were updated and modified very frequently. Moreover, the translation validation approach sounded more promising at this point. As a consequence, we did not maintain the invariant checker for a long period, and did not attack the certification of large programs.

Chapter 11

Proof of Semantic Equivalence

We focus on the automated proof of equivalence between source and compiled programs. The principle to prove the functional correctness of the compiled program with respect to the source code, by checking the equivalence of *local* computation steps at a given level of abstraction.

We discuss the issues inherent in the certification of the equivalence between source and compiled programs in Section 11.1. We formalize the translation validation technique [PSS98] in the framework, which we set up in Chapter 9 in Section 11.2.

Then, we prove this technique adapted for the proof of safety of compiled programs in Section 11.3. Indeed, when the proof of equivalence succeeds for some abstraction, then any *more abstract* invariant can be translated safely.

We provide implementation results showing the scalability of the method in Section 11.4. In the difference to other tools, our prototype does not input intermediate representations but rough source and compiled programs (so that the whole compilation is certified).

11.1 Principle and Related Work

We pointed out in Section 9.1 that a the verification of the *functional correctness* of the compile code was a very challenging and important goal in certified compilation. Indeed, it is particularly important to be sure that the compiled program indeed does what the source code says it should. More precisely, we wish to check that a trace of the compiled program corresponds to a trace of the source program and vice-versa, as in Definition 9.3.1.

Otherwise, we may encounter various kind of (possibly dramatic) errors: either misfunctioning due to the wrong implementation of the functions defined in the source code, or runtime errors due to a flawed translation.

Moreover, the proof of equivalence between source and compiled programs can be considered a strong documentation for the assembly code: indeed, it should describe what memory location corresponds to what source variable and prove this correspondence correct. As such, it can be used for the certification of critical compiled programs, e.g. in embedded systems and in aeronautics [TCoA99].

Several solutions to this issue can be found in the literature.

Theorem proving: A first approach consists in proving the compiler formally, with the help of a proof assistant.

It has been applied successfully to simple "toy" compilers, e.g. in [Ber98]. More recently, [Str02] presented a formal proof of correctness of a compiler for a subset of Java Card: this proof was the result of an extensive formalization of Java and on the verification of the compiler inside the Isabelle/HOL theorem proving environment [Pau94]. Currently, the Concert project led in the Inria aims at proving a fully functional, (moderately) optimizing C compiler; no publication is currently available about this ongoing project, but information can be found at `http://www-sop.inria.fr/lemme/concert/`.

Of course, such techniques are relevant only when the code of the compiler is publicly available. Moreover, this approach tend to be costly: not only the formal proofs tend to be long and not completely automated but also, a change in the code of the compiler may require part of the proof to be rewritten.

Translation validation: A second solution proceeds by performing the equivalence proof on a per-program basis: any time a program is compiled, it should be checked.

This approach, known as translation validation, was introduced by [PSS98]: this work focused on a synchronous compiler for the Signal language [ABG95]. Another similar tool was described in [Nec00]; the approach followed in this work is based on the checking of phases of an optimizing compilation, based on the RTL intermediate representation. This technique was also employed in [ZPFG02, ZPF$^+$02] so as to certify an Intel compiler for the Intel Itanium.

We also implemented a prototype based on the principles of translation validation in [Riv04b]: the certification of compilation should be done for every critical program. However, our tool is not based on any intermediate representation (it inputs source and assembly programs). Moreover, no knowledge about the way the compiler works is assumed, except that it should produce standard debugging information.

Invariant translation: A last solution is to resort the Invariant translation methods, which we described in Chapter 10.

Indeed, in case the invariant checking defined in Section 10.3 succeeds, then the property corresponding to the invariant is proved sound, independently from any assumption about the compiler. As a consequence, it proves that the compilation preserves some property of the source program in the compiled program. Obviously, the property preserved is not strong enough for our goal: it does not show that the assembly program implements correctly functions in the source code.

11.2 Design of a Translation Validation Procedure

In this section, we focus on the formalization and on the proof of the approach. In the end, we also relate our implementation results.

11.2.1 Formalization and soundness of the approach

The intuition behind translation validation is rather simple: if the source and the compiled program are "locally" equivalent, then, we can prove them globally equivalent. The purpose of this section is to state the local equivalence and to show the soundness.

Notations, and assumptions: In this section, we consider a source program P_{s}, and a compiled programs P_{c}. We use the same notations in the previous chapter. In particular, we define as usual:

- restricted sets of memory locations $\overline{\mathbb{X}}_{\mathrm{s}} \subseteq \mathbb{X}_{\mathrm{s}}$, and $\overline{\mathbb{X}}_{\mathrm{c}} \subseteq \mathbb{X}_{\mathrm{c}}$;
- restricted sets of control states $\overline{\mathbb{L}}_{\mathrm{s}} \subseteq \mathbb{L}_{\mathrm{s}}$, and $\overline{\mathbb{L}}_{\mathrm{c}} \subseteq \mathbb{L}_{\mathrm{c}}$;
- a mapping of memory locations $\Pi_{\mathbb{X}} : \overline{\mathbb{X}}_{\mathrm{s}} \to \overline{\mathbb{X}}_{\mathrm{c}}$;
- a mapping of control states $\Pi_{\mathbb{L}} : \overline{\mathbb{L}}_{\mathrm{s}} \to \overline{\mathbb{L}}_{\mathrm{c}}$;
- reduced programs $P_{\mathrm{s}}^{\mathrm{r}}$ and $P_{\mathrm{c}}^{\mathrm{r}}$, defined by the above restricted sets.

However, we *do not assume* that the compilation of P_{s} into P_{c} is sound. Indeed: our purpose is to state some conditions and to prove that the compilation of P_{s} into P_{c} is sound under this assumption.

We also require the restricted programs to be defined by table of symbolic transfer functions: if $l_s, l_s' \in \overline{\mathbb{L}}_{\mathrm{s}}$, then $\delta_{l_s, l_s'}$ denotes the transfer function describing the one-step transitions from l_s to l_s'.

Last, we require $l_{\mathrm{s}}^{\mathrm{i}}$ (resp. $l_{\mathrm{c}}^{\mathrm{i}}$) to be the entry point of $P_{\mathrm{s}}^{\mathrm{r}}$ (resp. $P_{\mathrm{c}}^{\mathrm{r}}$), and that $\Pi_{\mathbb{L}}(l_{\mathrm{s}}^{\mathrm{i}}) = l_{\mathrm{c}}^{\mathrm{i}}$ (as in Section 9.3.3).

Local equivalence: We now set up the "local equivalence" property; intuitively, it states that one step in the source restricted program should correspond to one step in the compiled, restricted program.

Definition 11.2.1. Local equivalence.

We say that the programs P_{s} and P_{c} are locally equivalent *with respect to $\Pi_{\mathbb{L}}$ and $\Pi_{\mathbb{X}}$ if and only if the following property holds:*

$$\begin{array}{c}\forall l_s, l_s' \in \overline{\mathbb{L}}_{\mathrm{s}},\ \forall \rho_c, \rho_c' \in \overline{\mathbb{M}}_{\mathrm{c}},\ \textit{if } l_c = \Pi_{\mathbb{L}}(l_s) \textit{ and } l_c' = \Pi_{\mathbb{L}}(l_s),\ \textit{then:}\\ (l_s, \rho_c \circ \Pi_{\mathbb{X}}) \rightarrow_{\mathrm{s}}^{\mathrm{r}} (l_s', \rho_c' \circ \Pi_{\mathbb{X}}) \Longleftrightarrow (l_c, \rho_c) \rightarrow_{\mathrm{c}}^{\mathrm{r}} (l_c', \rho_c')\end{array}$$

This property can also be stated in terms of symbolic transfer functions:

$$\begin{array}{c}\forall l_s, l_s' \in \overline{\mathbb{L}}_{\mathrm{s}},\ \forall \rho_c \in \overline{\mathbb{M}}_{\mathrm{c}},\ \textit{if } l_c = \Pi_{\mathbb{L}}(l_s) \textit{ and } l_c' = \Pi_{\mathbb{L}}(l_s),\ \textit{then:}\\ [\![\delta_{l_s, l_s'}]\!](\rho_c \circ \Pi_{\mathbb{X}}) = [\![\delta_{l_c, l_c'}]\!](\rho_c) \circ \Pi_{\mathbb{X}}\end{array}$$

A global way of stating local equivalence: At this point, we can provide another way of stating the local equivalence, which is based on a *global* statement.

We write $F^{\mathrm{r}}_{\mathrm{s}}$ (resp. $F^{\mathrm{r}}_{\mathrm{c}}$) for the semantic function of the restricted source program (resp. of the restricted compiled program); we recall that $F^{\mathrm{r}}_{\mathrm{s}}$ is defined by:

$$\begin{array}{rccl} F^{\mathrm{r}}_{\mathrm{s}}: & \mathcal{P}(\overline{\Sigma}_{\mathrm{s}}) & \longrightarrow & \mathcal{P}(\overline{\Sigma}_{\mathrm{s}}) \\ & \mathcal{E} & \mapsto & \mathcal{E} \cup \{\langle s_0, \ldots, s_n, s_{n+1}\rangle \mid \langle s_0, \ldots, s_n\rangle \in \mathcal{E} \wedge s_n \rightarrow^{\mathrm{r}}_{\mathrm{s}} s_{n+1}\} \end{array}$$

Moreover, if we let $\overline{\mathbb{S}}^{\mathrm{i}}_{\mathrm{s}} = \{\langle(\iota^{\mathrm{i}}_{\mathrm{s}}, \rho)\rangle \mid \rho \in \overline{\mathbb{M}}_{\mathrm{s}}\}$, the semantics of $P^{\mathrm{r}}_{\mathrm{s}}$ is defined by $[\![P^{\mathrm{r}}_{\mathrm{s}}]\!] = \mathbf{lfp}_{\overline{\mathbb{S}}^{\mathrm{i}}_{\mathrm{s}}} F^{\mathrm{r}}_{\mathrm{s}}$. Of course, the same properties and notations hold for the compiled program.

Lemma 11.2.1. Local equivalence, global formula.

The programs P_{s} and P_{c} are locally equivalent if and only if:

$$\forall \mathcal{E} \in \mathcal{P}(\overline{\Sigma}_{\mathrm{s}}),\ \Pi_{\Sigma}(F^{\mathrm{r}}_{\mathrm{s}}(\mathcal{E})) = F^{\mathrm{r}}_{\mathrm{c}}(\Pi_{\Sigma}(\mathcal{E}))$$

Proof.

Implication $\Rightarrow$: Let us assume that P_{s} and P_{c} are locally equivalent with respect to $\Pi_{\mathbb{L}}$ and $\Pi_{\mathbb{X}}$.
Let $\mathcal{E} \in \mathcal{P}(\Sigma)$. We assume that $\mathcal{E}$ is a singleton $\mathcal{E} = \{\sigma\}$ for some trace σ. Let us write $\sigma = \langle s^{\mathrm{s}}_0, \ldots, s^{\mathrm{s}}_n\rangle$, and $\Pi_{\Sigma}(\sigma) = \langle s^{\mathrm{c}}_0, \ldots, s^{\mathrm{c}}_n\rangle$, where $\forall i,\ s^{\mathrm{c}}_i = \Pi_{\mathbb{M}}(s^{\mathrm{s}}_i)$. Then:
- $\Pi_{\Sigma}(F^{\mathrm{r}}_{\mathrm{s}}(\mathcal{E})) = \{\langle s^{\mathrm{s}}_0, \ldots, s^{\mathrm{s}}_n, s^{\mathrm{s}}_{n+1}\rangle \mid s^{\mathrm{s}}_n \rightarrow^{\mathrm{r}}_{\mathrm{s}} s^{\mathrm{s}}_{n+1}\}$;
- $\Pi_{\Sigma}(F^{\mathrm{r}}_{\mathrm{s}}(\mathcal{E})) = \{\langle s^{\mathrm{c}}_0, \ldots, s^{\mathrm{c}}_n, s^{\mathrm{c}}_{n+1}\rangle \mid s^{\mathrm{c}}_n \rightarrow^{\mathrm{r}}_{\mathrm{s}} s^{\mathrm{c}}_{n+1}\}$

Moreover, the local equivalence entails that:

$$\forall s^{\mathrm{s}}_{n+1} \in \overline{\mathbb{S}}_{\mathrm{s}},\ \forall s^{\mathrm{c}}_{n+1} \in \overline{\mathbb{S}}_{\mathrm{c}},\ s^{\mathrm{s}}_n \rightarrow^{\mathrm{r}}_{\mathrm{s}} s^{\mathrm{s}}_{n+1} \iff s^{\mathrm{c}}_n \rightarrow^{\mathrm{r}}_{\mathrm{s}} s^{\mathrm{c}}_{n+1}$$

Therefore,

$$\Pi_{\Sigma}(F^{\mathrm{r}}_{\mathrm{s}}(\mathcal{E})) = F^{\mathrm{r}}_{\mathrm{c}}(\Pi_{\Sigma}(\mathcal{E}))$$

The results for any set $\mathcal{E} \in \mathcal{P}(\Sigma)$ follows from the case of singletons, since Π_{Σ}, $F^{\mathrm{r}}_{\mathrm{s}}$, and $F^{\mathrm{r}}_{\mathrm{c}}$ are continuous. Indeed, if $\mathcal{E} \subseteq \overline{\Sigma}_{\mathrm{s}}$, then

$$\begin{array}{rcll} \Pi_{\Sigma}(F^{\mathrm{r}}_{\mathrm{s}}(\mathcal{E})) & = & \Pi_{\Sigma}(F^{\mathrm{r}}_{\mathrm{s}}(\bigcup\{\{\sigma\} \mid \sigma \in \mathcal{E}\})) & \\ & = & \Pi_{\Sigma}(\bigcup\{F^{\mathrm{r}}_{\mathrm{s}}(\{\sigma\}) \mid \sigma \in \mathcal{E}\}) & \text{since } F^{\mathrm{r}}_{\mathrm{s}} \text{ is continuous} \\ & = & \bigcup\{\Pi_{\Sigma}(F^{\mathrm{r}}_{\mathrm{s}}(\{\sigma\})) \mid \sigma \in \mathcal{E}\} & \text{since } \Pi_{\Sigma} \text{ is continuous} \\ & = & \bigcup\{F^{\mathrm{r}}_{\mathrm{c}}(\Pi_{\Sigma}(\{\sigma\})) \mid \sigma \in \mathcal{E}\} & \text{as shown above} \\ & = & \ldots & \text{since } \Pi_{\Sigma}, F^{\mathrm{r}}_{\mathrm{c}} \text{ are continuous} \\ & = & F^{\mathrm{r}}_{\mathrm{c}}(\Pi_{\Sigma}(\bigcup\{\{\sigma\} \mid \sigma \in \mathcal{E}\})) & \\ & = & F^{\mathrm{r}}_{\mathrm{c}}(\Pi_{\Sigma}(\mathcal{E})) & \end{array}$$

The "global statement" for local equivalence follows.

Implication $\Leftarrow$: We assume that the "global statement" for local equivalence holds and establish the local one.
Let $l_s, l_s' \in \overline{\mathbb{L}}_{\rm s}$, $l_c = \Pi_{\mathbb{L}}(l_s)$, $l_c' = \Pi_{\mathbb{L}}(l_s')$, $\rho_c, \rho_c' \in \overline{\mathbb{M}}_{\rm c}$. We write $\rho_s = \rho_c \circ \Pi_{\mathbb{X}}$ and $\rho_s' = \rho_c' \circ \Pi_{\mathbb{X}}$. We assume that $(l_s, \rho_s) \to_{\rm s}^{\rm r} (l_s', \rho_s')$.
We let $\mathcal{E} = \{\langle (l_s, \rho_s) \rangle\}$. We know that:
- $\Pi_\Sigma(F_{\rm s}^{\rm r}(\mathcal{E})) = F_{\rm c}^{\rm r}(\Pi_\Sigma(\mathcal{E}))$;
- $\Pi_\Sigma(\mathcal{E}) = \{\langle (l_c, \rho_c) \rangle\}$;
- $\langle (l_s, \rho_s), (l_s', \rho_s') \rangle \in F_{\rm s}^{\rm r}(\mathcal{E})$, so that $\langle (l_c, \rho_c), (l_c', \rho_c') \rangle \in \Pi_\Sigma(F_{\rm s}^{\rm r}(\mathcal{E})) = F_{\rm c}^{\rm r}(\Pi_\Sigma(\mathcal{E}))$.

At this point, expanding the definition of $F_{\rm c}^{\rm r}$ allows us to derive the result:

$$(l_c, \rho_c) \to_{\rm c}^{\rm r} (l_c', \rho_c')$$

As a conclusion, the "local" statement for local equivalence, which we gave in Definition 11.2.1 holds. □

Soundness: We now state and prove the soundness of translation validation:

Theorem 11.2.2. Soundness of translation validation.
If $P_{\rm s}$ and $P_{\rm c}$ are locally equivalent with respect to $\Pi_{\mathbb{L}}$ and $\Pi_{\mathbb{X}}$, then the compilation of $P_{\rm s}$ into $P_{\rm c}$ is correct, with respect to the same mappings, i.e. :

$$[\![P_{\rm s}^{\rm r}]\!] \overset{\Pi_\Sigma}{\simeq} [\![P_{\rm c}^{\rm r}]\!]$$

Proof.

We assume that $P_{\rm s}$ and $P_{\rm c}$ are locally equivalent.
Then, The result follows directly from Lemma 11.2.1 and from a fixpoint transfer theorem like Theorem 2.3.2. Indeed:
- $\Pi_\Sigma(\overline{\mathbb{S}}_{\rm s}^{\rm i}) = \overline{\mathbb{S}}_{\rm c}^{\rm i}$;
- $\Pi_\Sigma(F_{\rm s}^{\rm r}(\mathcal{E})) = F_{\rm c}^{\rm r}(\Pi_\Sigma(\mathcal{E}))$.

As a consequence, a straightforward induction proves that

$$\Pi_\Sigma(\mathbf{lfp}_{\overline{\mathbb{S}}_{\rm s}^{\rm i}} F_{\rm s}^{\rm r}) = \mathbf{lfp}_{\overline{\mathbb{S}}_{\rm c}^{\rm i}} F_{\rm c}^{\rm r}$$

i.e.,

$$\Pi_\Sigma([\![P_{\rm s}^{\rm r}]\!]) = [\![P_{\rm c}^{\rm r}]\!]$$

Moreover, Π_Σ is clearly a bijection.
This proves the correctness of compilation of $P_{\rm s}$ into $P_{\rm c}$, with respect to the mappings $\Pi_{\mathbb{L}}$ and $\Pi_{\mathbb{X}}$ as stated in Theorem 9.3.3. □

The technique stated in the theorem above can be applied straightforwardly to the program considered in Section 9.3.1.

Definition 11.2.1. Translation validation.

Let P_s (resp. P_c) denote the source (resp. compiled) program introduced in Figure 9.3(a) (resp. Figure 9.3(b)). The mappings $\Pi_\mathbb{L}$ and $\Pi_\mathbb{X}$ for this pair of programs was given in Figure 9.4.
The symbolic transfer functions for the the restricted program P_s^r (resp. P_c^r) were given in Example 9.3.3 (resp. in Example 9.3.4).
These tables of transfer functions obviously satisfy the local equivalence property stated in Definition 11.2.1.
For instance, let us check the transitions between point l_4^s and l_5^s: these points of the reduced source program correspond to the control states l_{11}^c and l_{15}^c in the reduced compiled program. Moreover:

$$\begin{array}{rcl} \delta_{l_4^s, l_5^s} & = & \lfloor x \leftarrow x + t[i] \rfloor \\ \delta_{l_{11}^\mathrm{c}, l_{15}^\mathrm{c}} & = & \lfloor \mathbf{M}[\underline{x}] \leftarrow \mathbf{M}[\underline{x}] + \mathbf{M}[\underline{t} + \mathbf{M}[\underline{i}]] \rfloor \end{array}$$

Since $\Pi_\mathbb{X}(x) = \mathbf{M}[\underline{x}]$ and $\Pi_\mathbb{X}(t[i]) = \mathbf{M}[\underline{t} + \mathbf{M}[\underline{i}]]$, the two symbolic transfer functions above describe the same transitions up to $\Pi_\mathbb{X}$.
The case of the other transitions is similar.
As a conclusion, P_s and P_c enjoy the local equivalence property.

Remark 11.2.1. Formal compiler proof.

We mentioned in Section 11.1 that theorem proving was a solution for establishing the correctness of the compiler once for all. Then, the proof of correctness should establish a result similar to the equivalence stated in Theorem 11.2.2 for all programs *P_s and P_c.*
However, we should distinguish several kinds of proofs:

- *proofs based on* ***big-steps*** *semantics, akin to denotational or relational semantics (Section 3.2) only focus on initial and final states; they do not tell much on the* local *behavior of programs;*
- *proofs based on* ***small-steps*** *semantics proceed by establishing some kind of local equivalence property, similar to the one introduced in Definition 11.2.1.*

Of course, the latter kind of proofs is more informative; indeed, such proofs provide a rather strong link between the source and the compiled programs.

11.2.2 Adapted decision procedure

The purpose of this section is to propose an algorithm for checking the local equivalence property defined in the previous section. The decision procedure should input a pair of symbolic transfer functions $(\delta^\mathrm{s}, \delta^\mathrm{c})$ (where δ^s describes a transition in P_s^r, and δ^c describes a transition in P_c^r), and attempts to prove that P_s^r and P_c^r are equivalent:

Definition 11.2.2. Symbolic transfer functions equivalence.

The functions δ^{s} and δ^{c} are equivalent *if and only if*

$$\forall \rho_s \in \overline{\mathbb{M}}_{\mathrm{s}},\ \rho_c \in \overline{\mathbb{M}}_{\mathrm{c}},\ \rho_c = \Pi_{\mathbb{M}}(\rho_s) \Longrightarrow \Pi_{\mathbb{M}}(\delta^{\mathrm{s}}(\rho_s)) = \delta^{\mathrm{c}}(\rho_c)$$

Obviously, checking the local equivalence property stated in Definition 11.2.1 reduces to proving the equivalence of symbolic transfer functions, in the sense of Definition 11.2.2.

Of course, we do not expect the decision procedure to be complete, since the equivalence of symbolic transfer functions is undecidable; therefore, it may fail to establish the equivalence of two equivalent symbolic transfer functions.

As usual, we expect the decision procedure to be sound: if it succeeds, then the two arguments should be equivalent in the sense of Definition 11.2.2.

The algorithm exposed here is much more simple than the algorithm which we effectively implement; indeed, this algorithm presents a very high asymptotic complexity, so that a rather tricky implementation is somewhat required in order to achieve fast decision procedures.

Our procedures may handle *assumptions* under the form of finite lists of boolean expressions (aka conditions); we write an assumption C (condition shall be denoted $c_0, \ldots$).

Equivalence of expressions: Before we describe the algorithm, we mention two important subroutines. The first one attempts to prove two expressions equivalent.

Definition 11.2.3. Expression equivalence.

Let e_0, e_1 be two expressions, with variables in $\overline{\mathbb{X}}_{\mathrm{c}}$ and C be an assumption. We say that e_0 and e_1 are equivalent under the assumption C if and only if, for all store $\rho \in \overline{\mathbb{M}}_{\mathrm{c}}$,

$$(\forall c_i \in C,\ [\![c_i]\!](\rho) = \mathbf{true}) \Longrightarrow [\![e_0]\!](\rho) = [\![e_1]\!](\rho)$$

If the procedure proves e_0 and e_1 equivalent, we write $C \vdash e_0 \sim e_1$.

In our implementation, a very large number of rules needed to be included in the decision procedures; we give a few examples here:

- Equality of constants: some constant (e.g., floating point constants) have several representations, including hexadecimal representations, standard representations.
- If $C \vdash e_0 \sim e_1$, then, for any expression e and any operator $\oplus$, $C \vdash e_0 \oplus e \sim e_1 \oplus e$.
- Faster implementations for comparisons: for instance, if e_0 is an integer expression, then testing whether $e_0 < 0$ amounts to checking whether the sign bit of the result of the evaluation of e_0 is 1 (this can be implemented with a shift right); this allows to test some simple conditions without using the condition register (less instructions), which we described in Section 9.2.1.

  Many other examples of faster operators can be found in practice.

incompatible branches	$\dfrac{C \vdash \mathbf{false}}{C \models \delta^{\mathrm{s}} \sim \delta^{\mathrm{c}}}\ \text{False}$
empty functions	$\dfrac{}{C \models \Box \sim \Box}\ \text{Empty}$
assignments	$\dfrac{C \vdash e_s \sim e_c \qquad \Pi_{\mathbb{X}}(x_s) = x_c}{C \models \lfloor x_s \leftarrow e_s \rfloor \sim \lfloor x_c \leftarrow e_c \rfloor}\ \text{Assign}$
n-ary assignments	generalizes the case of unary assignments
conditional (1)	$\dfrac{C; e \models \delta^{\mathrm{s}}_t \sim \delta^{\mathrm{c}} \qquad C; \neg e \models \delta^{\mathrm{s}}_f \sim \delta^{\mathrm{c}}}{C \models \lfloor e\ ?\ \delta^{\mathrm{s}}_t \mid \delta^{\mathrm{s}}_f \rfloor \sim \delta^{\mathrm{c}}}\ \text{If}_{\mathbf{l}}$
conditional (2)	$\dfrac{C; e \models \delta^{\mathrm{s}} \sim \delta^{\mathrm{c}}_t \qquad C; \neg e \models \delta^{\mathrm{s}} \sim \delta^{\mathrm{c}}_f}{C \models \delta^{\mathrm{s}} \sim \lfloor e\ ?\ \delta^{\mathrm{c}}_t \mid \delta^{\mathrm{c}}_f \rfloor}\ \text{If}_{\mathbf{r}}$

Figure 11.1: Decision procedure

In practice, we are interested with the case where e_0 is an expression based on *source variables* and e_1 is an expression based on *assembly memory locations*. This case reduce to the previous case. Indeed, we can substitute source variables with assembly memory locations in e_0, thanks to $\Pi_{\mathbb{X}}$; we write $\Pi_{\mathbb{X}}(e_0)$ for the resulting expression. Then, we can apply the regular decision procedure to $(\Pi_{\mathbb{X}}(e_0), e_1)$.

Search for contradiction in set of hypotheses: The second important function attempts to find a contradiction among a set of hypotheses. If this procedure succeeds on an assumption C, we write $C \vdash \mathbf{false}$. The soundness of this procedure states, that if $C \vdash \mathbf{false}$, then:

$$\{\rho \in \overline{\mathbb{M}}_{\mathrm{c}} \mid \forall c_i \in C,\ [\![c_i]\!](\rho) = \mathbf{true}\} = \emptyset$$

Among the examples of rules for this decision procedure, we can cite the standard rules below:

$$\frac{}{C; e_0 < e_1; e_0 \geq e_1 \vdash \mathbf{false}} \qquad\qquad \frac{}{C; b; \neg b \vdash \mathbf{false}}$$

Equivalence of symbolic transfer functions: We write $C \models \delta^{\mathrm{s}} \sim \delta^{\mathrm{c}}$ if the procedure succeeds in proving the equivalence of δ^{s} and δ^{c}. The algorithm of the decision procedure is described as a set of rules in Figure 11.1.

This decision procedure can be proved sound:

Theorem 11.2.3. Equivalence of symbolic transfer functions.

Let C be an assumption, and $(\delta^{\mathrm{s}}, \delta^{\mathrm{c}})$ be a pair of symbolic transfer functions. We assume that $C \models \delta^{\mathrm{s}} \sim \delta^{\mathrm{c}}$. Then:

$$\forall \rho_s \in \overline{\mathbb{M}}_{\mathrm{s}},\ \rho_c \in \overline{\mathbb{M}}_{\mathrm{c}},\ \rho_c = \Pi_{\mathbb{M}}(\rho_s) \wedge (\forall c_i \in C,\ [\![c_i]\!](\rho) = \mathbf{true}) \Longrightarrow \Pi_{\mathbb{M}}(\delta^{\mathrm{s}}(\rho_s)) = \delta^{\mathrm{c}}(\rho_c)$$

In particular, if C is empty, then δ^{s} and δ^{c} are equivalent in the sense of Definition 11.2.3.

Proof.

By induction on the proof trees for deriving $C \models \delta^{\mathrm{s}} \sim \delta^{\mathrm{c}}$. □

Definition 11.2.2. Equivalence of symbolic transfer functions.

The rules presented in Figure 11.1 allow to prove the equivalence of all transfer functions involved in Example 11.2.1, in a very straightforward manner.

Implementation: The decision procedure inputs two symbolic transfer functions and attempts to prove their equivalence, by applying the rules from the conclusion to the premises and close each branch in the proof with either of rules False, Empty and Assign. All rules in Figure 11.1 but the False rule are guided by the nature of the transfer functions on both sides. In practice, the decision procedure should attempt to derive contradictions among the hypotheses, so as to apply the rule False as soon as possible.

11.2.3 Issues with the computation of the reduced programs

We discussed the issue of optimizations in Section 9.4, and showed that alternate forms of program reductions should be used when the compiler performs optimizations.

When the assembly code is optimized, the algorithm for translation validation should be applied to the reduced programs defined in Section 9.4. When it succeeds, it proves the correctness of the compilation with respect to the mappings used for the computation of the reduced programs.

As a consequence, the main difference in the translation validator lies in the assembly and source front-ends, since they should also compute the transfer functions for the reduced source and assembly programs. We described this technique and provided some implementation results in [Riv04b].

11.3 Application to Invariant Translation

In this section, we study the link between the translation validation and the invariant checking procedure, which we introduced in Section 10.3.

11.3.1 Soundness of the approach

First of all, we show that, if the translation validation succeeds, then any invariant for the source, restricted program can be translated safely.

Therefore, we assume that the same conditions as in Section 10.2 are fulfilled. We use the same notations as well; in particular, we assume that a sound abstract invariant $\mathfrak{I}_{\mathrm{s}} \in \mathbb{L}_{\mathrm{s}} \to D^{\sharp}_{\mathbb{M}}$ for $P^{\mathrm{r}}_{\mathrm{s}}$ is given.

Theorem 11.3.1. Invariant translation justification.

Let us assume that the test of local equivalence of P_{s} and P_{c} succeeds and that $\mathfrak{I}_{\mathrm{s}}$ is a sound invariant for P_{s}.

Then, the invariant $\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}$ defined as in Section 10.2.1 provides a sound approximation for the semantics of the reduced, compiled program:

$$\forall \langle \ldots, (l_c, \rho_c)\rangle \in [\![P^{\mathrm{r}}_{\mathrm{c}}]\!],\ \rho_c \in (\Pi_{\mathbb{X}})^{-1} \circ \gamma^{\sharp}_{\mathbb{M},\mathrm{s}}(\mathfrak{I}^{\mathrm{r}}_{\mathrm{c}}((\Pi_{\mathbb{L}})^{-1}(l_c)))$$

Proof.

Follows from Theorem 10.2.1 and Theorem 11.2.2. □

As a consequence, $\mathfrak{I}_{\mathrm{s}}$ can be used as the basis for computing an invariant $\mathfrak{I}_{\mathrm{c}}$ for the whole compiled program P_{c}, as done in Section 10.2.2.

11.3.2 Comparison with invariant checking

The result, which we provided in the last subsection shows that the approach consisting in performing a proof of equivalence and then an invariant translation somewhat turns out to be a substitute for the invariant translation and invariant checking. Indeed, the translation validation proves the equivalence of the restricted semantics of the source and compiled program, so that the correctness of the translated invariant does not depend on the correctness of compilation anymore.

Invariant checking and translation validation are two ways to check an identity among least-fixpoint formulas:

- invariant checking performs a *global* fixpoint checking, by verifying local soundness conditions; it is specific to some abstract domain;
- translation validation relies on the *local* checking of the hypotheses of a fixpoint transfer theorem; it is not specific to any abstraction.

Translation validation presents several advantages here:

- it proves a stronger property: as shown above, the soundness of the translated invariant can be deduced from translation validation, but the converse does not hold (the success of invariant checking does not prove the compilation correct);
- it is not specific to any abstract domain, but the program reduction;

- by contrast, the invariant checking procedure consists in an abstract interpreter, which should be precise enough to:
  - validate invariants produced by a source analyzer (which means that it should be at least as precise as the source analyzer used to synthesize $\mathfrak{I}_{\mathrm{s}}$);
  - deal with the specific features of the assembly languages, such as the issues mentioned in Section 10.3.2.
- the practical cost of translation validation turns out rather reasonable, as the benchmarks, which provide in Section 11.4 prove; on the contrary, the implementation of invariant checking turned out involved and the resulting performances disappointing, as we pointed out in Section 10.3.3.

The better efficiency of the translation validation stems from the fact that it allows for equivalent, more simple expressions to be recognized among the symbolic transfer functions for the source program. Indeed, the decision procedure described in Section 11.2.2 can be enhanced so as to prove simplified assembly transfer functions when it succeeds in proving the equivalence. For instance, this possibility turns out particularly useful in the following case:

Definition 11.3.1. Conversion of a short integer into a floating point.

Let us consider the compilation of the source statement $f = \mathbf{cast}_{\mathrm{short}\rightarrow\mathrm{float}} i$. *As we can remark, the computations involved in the conversions are quite involved. Figure 11.2 shows the corresponding sequence of assembly code produced by* **gcc**. *In this example, we explain how this algorithm works. First, note that the floating point registers store* 64-bit *floating points (i.e., values of type double).*

We recall that the most common floating point data-types are respectively 32 bits ("float" C type) and 64 bits ("double" C type) long. Moreover, a floating point value is made of

- *a* sign bit *s;*
- *an* exponent *e (8 bits for float, 11 bits for double) decremented with a* bias *b (respectively 127 and 1023);*
- *and a* mantissa *m of n bits (*$n = 23$ *for float,* $n = 52$ *for double).*

The value corresponding to such a floating point number is $2^{e-b} \cdot (1+2^{-n} \cdot m)$ *(the mantissa represents a fraction in the range* $[0, 1]$*). For more details about the representation of floating point numbers, we refer to [CS85].*

We can summarize the conversion algorithm displayed in Figure 11.2 as follows:

- *i is represented as a 32-bit integer (it was originally a 16-bit integers), thanks to the instruction* `extsh` *;*
- *the* `xoris` *instruction flips the highest bit in i;*
- *then, a bit-bield made of an hexadecimal constant and of i is formed and loaded into a 64-bit floating point register; the value of this result is* $2^{52} + 2^{31} + i$, *as a double;*
- *the constant* $C_0 = 2^{52} + 2^{31}$ *is loaded into another floating point register;*
- *the difference is computed (* `fsub` *), so that we get i expressed as a double;*
- *last the* `frsp` *instruction rounds the double of value x into a 32-bit floating point value (this rounding does not change the value, it merely modifies the internal rep-*

`lis 11,f@ha`	load address of f
`lis 9,i@ha`	load i
`lhz 0,i@l(9)`	$\mathbf{gpr}_0 \leftarrow i$
`extsh 0,0`	sign extension
`lis 9,0x4330`	$\mathbf{gpr}_9 \leftarrow$ `0x4330 0000 0000 0000`
`lis 10,.LC0@ha`	
`la 10,.LC0@l(10)`	$\mathbf{gpr}_{10} \leftarrow \underline{C_0}$
`lfd 13,0(10)`	$\mathbf{fpr}_{13} \leftarrow$ `0x4330 0000 0000 0000 1111 1 ... 1`
`xoris 0,0,0x8000`	$\mathbf{gpr}_0 \leftarrow \mathbf{gpr}_0 \oplus \mathbf{gpr}_0$
`stw 0,12(31)`	
`stw 9,8(31)`	
`lfd 0,8(31)`	$\mathbf{fpr}_0 \leftarrow \langle \mathbf{gpr}_9 \mid \mathbf{gpr}_0 \rangle$
`fsub 0,0,13`	$\mathbf{fpr}_0 \leftarrow \mathbf{fpr}_0 - \mathbf{fpr}_{13}$
`frsp 0,0`	rounding into a floating point value
`stfs 0,f@l(11)`	store result in f

Figure 11.2: Conversion of a short integer into a floating point

resentation in order to comply with the floating point representation).
The translation validation decision procedure recognizes some sub-expressions as conversions, when a conversion appears in the source expression. Moreover, it can produce a simplified transfer function, containing a mere type conversion. This higher level operation would be more amenable to static analysis, e.g., in the invariant propagation (Section 10.2.2).
By contrast, a satisfactory handling of such sequences of assembly instructions in the invariant checking would require an abstract domain to be designed so as to collect expressions and allow other domains to use them; this would make the design of the invariant checker tedious.

The issue described in Example 11.3.1 did not arise in the program $\mathcal{P}_1^1$ considered in Section 10.3.3, so that we came across this problem after applying translation validation to larger programs. This increased our confidence in the adequation of translation validation to our goal.

11.3.3 On the need for invariant translation and safety checking

Let us assume that the correctness of the compilation of P_{s} into P_{c} can be proved by translation validation, and that we wish to prove that P_{c} is safe. Moreover, we assume that the analysis of P_{s} proves it safe.

The translation of source invariants may seem useless, since the source and the assembly program are proved equivalent and the source program is also proved safe.

However, we may still be interested in performing it, so as to get a better guarantee of safety of the compiled program. Indeed, the definition of the runtime errors may be more natural at the assembly level, so that the verification of the safety conditions using the translated invariants can still be useful. We chose to perform it in the implementation, which we describe in Section 11.4 (it was also a great opportunity to compare the approach based on translation validation and the approach based on invariant checking in practice).

11.4 Application to real software

We implemented this approach in OCaml [OCa] in 2003, and checked its ability to scale up. We reported about this prototype in [Riv04b]. Our tool performed an Invariant translation preceded by a Translation validation step which allows to deal with simplified assembly symbolic transfer functions when translating invariants and to avoid coping with abstract invariant checking, since the latter technique generated disappointing results in Section 10.3.3.

Implementation: Our goal was to certify automatically both the compilation and the absence of runtime errors (RTE) in the compiled assembly programs. The target architecture is a 32 bits version of the Power-PC processor; the compiler is **gcc** 3.0.2 for Embedded ABI (cross-compiler). The source invariants are computed by the ASTRÉE analyzer [BCC+03a] (we give more details about ASTRÉE in Section 5.1) and achieve a very low false alarms number when used for checking RTE.

This prototype was aimed at validating the compilation of programs of the first family of embedded applications presented in Section 5.3.3. We describe the main characteristics of these programs in Section 5.1.1.

The translation validator handles most C features (excluding dynamic memory allocation through pointers which is not used in the family of highly critical programs under consideration): procedures and functions, structs, enums, arrays and basic data-types and all the operations on these data-types. The fragments of Power-PC assembly language handled by our implementation is ways larger than the fragment described in Section 9.2 as well. In particular, a restricted form of alias is needed so as to validate the passing by reference of some function arguments like arrays. Non-determinism is also accommodated (volatile variables). The mappings $\Pi_{\mathbb{X}}$ (for variables) and $\Pi_{\mathbb{L}}$ (for program points) are extracted from standard debugging information. The verifier uses the Stabs format (hence, it inputs assembly programs including these data), in the same way as the invariant translator and invariant checking described in Section 10.3.3.

The decision procedure involved in the translation validation is based on the same principle as the decision procedure described in Section 11.2.2, even though the implementation is particularly tricky so as to keep the cost down. Moreover, it required two very simple analyses to be performed, so as to collect additional assumptions:

- an analysis collecting equality relations;

• a congruence analysis [Gra89], so as to check the correctness of memory accesses.

Moreover, the verification of the soundness of the assembly code requires only interval constraints to be translated: other constraints do not need to be translated, which makes the invariant translation much more efficient and simple.

The whole development amounts to about 33 000 lines of OCaml code: The various parsers and interfaces (e.g. with the source analyzer) are about 17 000 lines; the kernel of the certifier (the implementation of the symbolic transfer functions and the prover) is about 6 000 lines; the symbolic encoding functions (i.e., the formal definition of the semantics of the source and assembly languages) are about 3 000 lines; the invariant translator and the certifier are about 5 000 lines. The most critical and complicated part of the system corresponds to the symbolic composition (2 000 lines) and to the prover (1 500 lines).

Benchmarks: The whole process was ran on a 2.4 GHz Intel Xeon with 4 Gbytes of RAM. Translation validation succeeds on the three programs: no alarm is raised; hence the compiled programs are proved equivalent to the source code. The results of the benchmarks are given in the table below (sizes are in lines, times in seconds).

Code	Size		Time					Alarms	
	Source (LOCs)	Assembly (LOCs)	Parsing C	Parsing Power-PC	Mapping building	Translation validation	Invariant translation	Trans. valid.	RTE
$\mathscr{P}_1^1$	370	1 930	0.04	0.08	0.03	0.14	0.23	0	0
$\mathscr{P}_2^1$	9 500	56 600	0.53	0.96	0.39	0.62	8.22	0	0
$\mathscr{P}_3^1$	70 000	344 000	2.97	13	0.81	9.45	84.5	0	0

Conclusions: First, we note that the translation validation succeeds (no alarm); as a consequence, the compilation is proved correct, and the invariant translation is also justified. Second, the translated invariant allows the certification of the compiled code. Note that [Riv04b] reported a few alarms in the case of $\mathscr{P}_3^1$: these were due to the use of a former version of ASTRÉE. The ASTRÉE analyzer was improved since this date and now allows to compute a more precise invariant for this program, achieving the result of 0 false alarm.

Last, we are pleased to note that this technique does scale up, in the difference to the implementation of invariant checking, which we described in Section 10.3.3.

Perspectives: At the time of the writing, we are working on a new, improved implementation, so as to replace the initial prototype. It focuses on the certification of *binaries*, instead of assembly code. In particular, we hope to improve the handling of debugging information and the decision procedure, (we envisage to make it safer, e.g., by letting it generate proof terms).

Chapter 12

Conclusion

In this thesis, we examined various abstractions for sets of traces and applied these to different applications, all related to the certification of safety critical embedded systems.

Let us review the main contribution and the opportunities of future work for the three main parts of the thesis.

12.1 Trace partitioning

We proposed a powerful, generic framework for trace partitioning and applied it to several practical problems. First, we derived and implemented a trace partitioning domain in ASTRÉE, which turned out to play a major role in the performance of the analyzer, both in time and in precision. Second, we designed a domain, which allows to state powerful properties of executions, and which is particularly helpful in the alarm investigation.

The most important area for future work in this part seems to be the improvement of the technique proposed in Chapter 6.

- First, we envisage to attempt to use this domain in order to prove significant functional properties of programs, with the help of the ASTRÉE static analyzer.
- Second, we could extend the automata-based abstraction (Section 6.3) with a widening operator.
- Last, we could also explore the idea of using an abstract system derived from the property of interest in order to guide the widening strategy.

12.2 Alarm investigation

We provided the basis for setting up semi-automatic alarm investigation techniques. In particular, we proposed various families of relevant slicing criteria and algorithms for extracting semantic slices of programs, which is a very important step, when considering very large programs, as is the case in the ASTRÉE project. Second, we formalized families of dependences adapted to the alarm investigation: first, we restrict to the dependences, which are observable in a semantic slice; second, we consider in priority the dependences which have significant chances to play a role in abnormal behaviors (such as the occurrence of large values in programs, and in abstract invariants). These methods were implemented in prototypes based on ASTRÉE, and showed positive early results.

Obviously, much work remains to be done, before we can implement an automatic alarm investigation module inside ASTRÉE :

- Automatize the synthesis of semantic slicing criteria, from the invariants generated by ASTRÉE, and from the results of the abstract dependence analyses.
- The synthesis of error scenarios should be automatized; for instance, we plan to investigate the generation of collections of constraints, so as to characterize inputs, which would *always* cause an error.
- Other kinds of abstract dependences should be studied, e.g., in order to consider more involved families of predicates.

12.3 Certified compilation

We set up a general formalization for compilation. We defined and formalized several certified compilation algorithms in this framework, including the invariant translation, the invariant checking and the translation validation techniques. We implemented and compared these methods; in the end, we conclude that the equivalence checking method (translation validation) presents many advantages over the invariant checking technique. Not only it verifies the correctness of compilation, but also it turns out more efficient (in time) than the invariant checking. Overall, the translation validation prototype was plainly successful in proving the correctness of the compilation of large applications in a reasonable amount of time.

At this point, we are working on the improvement of the translation validation prototype, which should be used in industrial certification processes. Another direction for future work consists in considering other programming paradigms, such as synchronous languages.

Bibliography

[Aba99] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, 1999.

[ABG95] P. Amagbégnon, L. Besnard, and P. Le Guernic. Implementation of the data-flow synchronous language SIGNAL. In *Conference on Programming Language Design and Implementation (PLDI'95)*, pages 163–173, La Jolla (USA), June 1995. ACM Press, New York.

[ABHR99] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *26th Symposium on Principles of Programming Languages (POPL'99)*, pages 147–160, San Antonio (USA), 1999. ACM Press, New York.

[Abr89] Jean-Raymond Abrial. A formal approach to large software construction. In *Mathematics of Program Construction (MPC)*, volume 375 of *LNCS*, pages 1–20, Groningen (The Netherlands), June 1989. Springer.

[AFMW96] M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache Behavior Prediction by Abstract Interpretation. In *3rd Static Analysis Symposium (SAS'96)*, volume 1145 of *LNCS*, pages 51–66, Aachen (Germany), September 1996. Springer.

[AL98] G. Ammons and J. R. Larus. Improving data-flow analysis with path profiles. In *Conference on Programming Languages, Design and Implementation (PLDI'98)*, pages 72–84, Montréal (Canada), November 1998. ACM Press, New York.

[ANS99] ANSI ISO/IEC. *International Standard – Programming Languages – C*, 1999.

[App99] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1999.

[App01] A. W. Appel. Foundational Proof-Carrying Code. In *16th Symposium on Logics in Computer Science (LICS'2001)*, pages 247–256, Boston (USA), June 2001. IEEE Computer Society Press.

[BCC$^+$02] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software, invited chapter. In T. Mogensen, D.A. Schmidt, and I.H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 85–108. Springer, October 2002.

[BCC$^+$03a] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A Static Analyzer for Large Safety Critical Software. In *Conference on Programming Languages, Design and Implementation (PLDI'03)*, pages 196–207, San Diego (USA), June 2003. ACM Press, New York.

[BCC$^+$03b] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. *User Manual of the* ASTRÉE *Static Analyzer*. ASTRÉE, 2003.

[Ber98] Y. Bertot. A certified compiler for an imperative language. Technical Report RR-3488, INRIA, 1998.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BGS94] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, 1994.

[BGS97] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 361–377, Zurich (Switzerland), September 1997.

[BL96] T. Ball and J. R. Larus. Efficient path profiling. In *IEEE/ACM International Symposium on Microarchitecture (MICRO 96)*, pages 46–57, Paris, France, December 1996.

[BMMR01] T. Ball, R. Majumdar, T. D. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Conference on Programming Languages, Design and Implementation (PLDI'01)*, pages 203–213, Snowbird (USA), June 2001. ACM Press, New York.

[BNR03] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. In *30th Symposium on Principles of Programming Languages (POPL'03)*, pages 97–105, New Orleans (USA), January 2003. ACM Press, New York.

[Bou93] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *International Conference on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 128–142, Novosibirsk, Russia, June 1993. Springer.

[BR01] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *8th International SPIN Workshop*, volume 2057 of *LNCS*, pages 103–122, Toronto (Canada), May 2001. Springer.

[BR04] G. Balakrishnan and T. W. Reps. Analyzing memory accesses in x86 executables. In *13th International Conference on Compiler Construction (CC'04)*, LNCS, pages 5–23, Barcelona (Spain), April 2004. Springer.

[Bry86] R.E. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[CBC93] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Symposium on Principles of Programming Languages (POPL'93)*, pages 232–245, Charleston (USA), January 1993. ACM Press, New York.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Symposium on Principles of Programming Languages (POPL'77)*, pages 238–252, Los Angeles, California, January 1977. ACM Press, New York, NY.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th Symposium on Principles of Programming Languages (POPL'79)*, pages 269–282, San Antonio, Texas, January 1979. ACM Press, New York, NY.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CC92b] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, August 1992.

[CC02] P. Cousot and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *29th Symposium on Principles of Programming Languages (POPL'02)*, pages 178–190, Portland, Oregon, January 2002. ACM Press, New York, NY.

[CCF$^+$05] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *European Symposium On Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 21–30, Edimburgh (Scotland), April 2005. Springer.

[CCL98] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. *Information and Software Technology*, 40(11-12):595–608, November 1998. Special issue on program slicing.

[CF89] R. Cartwright and M. Felleisen. The Semantics of Program dependence. In *Conference on Programming Languages, Design and Implementation (PLDI'89)*, pages 13–27, Portland (USA), June 1989. ACM Press, New York.

[CGJ$^+$00] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *12th Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *LNCS*, pages 154–169, Chicago (USA), July 2000. Springer.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th Symposium on Principles of Programming Languages (POPL'78)*, pages 84–97, Tucson, Arizona (USA), January 1978. ACM Press, New York.

[CL96] C. Colby and P. Lee. Trace-Based Program Analysis. In *23rd Symposium on Principles of Programming Languages (POPL'96)*, pages 195–207, St. Petersburg Beach, (USA), January 1996. ACM Press, New York.

[CL05] B.-Y. Chang and R. Leino. Abstract interpretation with alien expressions and heap structures. In *4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 147–163, Paris (France), January 2005. Springer.

[Col96] C. Colby. *Semantics-based Program Analysis via Symbolic Composition of Transfer Relations.* PhD thesis, Carnige Mellon Univeristy, August 1996.

[Cou78] P. Cousot. *Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes.* PhD thesis, Université de Grenoble, 1978.

[Cou81] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pages 303–342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[Cou97a] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997.

[Cou97b] P. Cousot. Types as abstract interpretations. In *24th Symposium on Principles of Programming Languages (POPL'97)*, pages 316–331, Paris, January 1997. ACM Press, New York.

[CS85] IEEE Computer Society. IEEE standard for binary floating-point arithmetic. Technical report, ANSI/IEEE Std 754-1985, 1985.

[DD77] D. Denning and P. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20(7):504–513, July 1977.

[Den76] D. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Conference on Programming Languages, Design and Implementation (PLDI'94)*, pages 230–241, Orlando (USA), June 1994. ACM Press, New York.

[Dji75] E. W. Djikstra. Guarded commands and formal derivation of programs. *Communications of the Association for Computer Machinery*, 18(8):453–457, August 1975.

[DLS02] M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *Conference on Programming Languages, Design and Implementation (PLDI'02)*, pages 57–68, Berlin (Germany), May 2002. ACM Press, New York.

[DRS03] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Conference on Programming Languages, Design and Implementation (PLDI'03)*, pages 155–167, San Diego (USA), June 2003. ACM Press, New York.

[DSC98] David Déharbe, Subash Shankar, and Edmund M. Clarke. Model checking vhdl with cv. In *Formal Methods in Computer-Aided Design (FMCAD'98)*, volume 1522 of *LNCS*, pages 508–514, Palo Alto (USA), November 1998. Springer.

[ea96] J. L. Lions et al. ARIANE 5, flight 501 failure, report by the inquiry board, 1996.

[EMCP02] O. Grumberg E. M. Clarke and D. Peled. *Model-Checking.* MIT Press, 2002.

[Ere04] G. Erez. Generating counter examples for sound abstract interpretation. Master's thesis, Tel Aviv University, 2004.

[FDHH04] C. Fox, S. Danicic, M. Harman, and R. M. Hierons. ConSIT: a fully automated conditioned program slicer. *Software - Practice and Experience*, 34(1):15–46, 2004.

[Fer04a] J. Feret. The arithmetic-geometric progression abstract domain. In *6th conference on Verification, Model-Cecking and Abstract Interpretation (VMCAI'05)*, volume 3385 of *LNCS*, pages 2–18, Paris (France), January 2004. Springer.

[Fer04b] J. Feret. Static analysis of digital filters. In *European Symposium On Programming (ESOP'04)*, number 2986 in LNCS, Barcelona (Spain), April 2004. Springer.

[FLL$^+$02] C. Flanagan, K. R. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Conference on Programming Languages, Design and Implementation (PLDI'02)*, pages 234–245, Berlin (Germany), May 2002. ACM Press, New York, NY.

[FMW97] C. Ferdinand, F. Martin, and R. Wilhelm. Applying Compiler Techniques to Cache Behavior Prediction. In *Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS)*, pages 37–46, Las Vegas (USA), June 1997. ACM Press, New York.

[GJJM03] F. Gaucher, E. Jahier, B. Jeannet, and F. Maraninchi. Automatic state reaching for debugging reactive programs. In *5th International Workshop on Automated Debugging (AADEBUG'03)*, Ghent (Belgium), September 2003.

[GM82] J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symp. on Security and Privacy*. IEEE Computer Society Press, 1982.

[GM03] R. Giacobazzi and I. Mastroeni. Non-standard semantics for program slicing. *Higher-Order and Symbolic Computation (HOSC)*, 16(4):297–339, 2003. Special issue on Partial Evalution and Semantics-Based Program Manipulation.

[GM04] R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *31st Symposium on Principles of Programming Languages (POPL'04)*, pages 186–197, Venice (Italy), janvier 2004. ACM Press, New York.

[GMJ$^+$02] D. Grossman, J. G. Morrisett, T. Jim, M. W. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Conference on Programming Languages, Design and Implementation (PLDI'02)*, pages 282–293, Berlin (Germany), May 2002. ACM Press, New York.

[Gra89] P. Granger. Static analysis of arithmetical congruences. In *International Journal of Computer Mathematics*, volume 30, pages 165–190, 1989.

[Gra92] P. Granger. Improving the results of static analyses programs by local decreasing iteration. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS'92)*, volume 652 of *LNCS*, pages 68–79. Springer, December 1992.

[GRS00] R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HDSS96] M. Harman, S. Danicic, Y. Sivagurunathan, and D. Simpson. The next 700 slicing criteria. In *2nd UK workshop on program comprehension*, Durham University (UK), July 1996.

[HDT87] S. Horwitz, A. J. Demers, and T. Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[HHF$^+$02] R. M. Hierons, M. Harman, C. Fox, L. Ouarbya, and M. Daoudi. Conditioned slicing supports partition testing. *Journal of Software Testing, Verification and Reliability.*, 12(1):23–28, 2002.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In *Algebraic Methodology and Software Technology (AMAST '93)*, Workshops in Computing, pages 83–96, Twente (Netherlands), June 1993. Springer.

[HR80] L. H. Holley and B. K. Rosen. Qualified data flow problems. In *7th Symposium on Principles of Programming Languages (POPL'80)*, pages 68–82, Las Vegas (Nevada), June 1980. ACM Press, New York.

[HRB88] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Conference on Programming Languages, Design and Implementation (PLDI'88)*, pages 35–46, Atlanta (USA), June 1988. ACM Press, New York.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing using Program Dependence Graphs. *ACM Transactions On Programming Languages And Systems (TOPLAS)*, 12(1):January, 26–60 1990.

[HT98] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *5th International Static Analysis Symposium (SAS'98)*, volume 1503 of *LNCS*, pages 200–214, Pisa (Italy), September 1998. Springer.

[Jea03] B. Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, 2003.

[JHR99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *6th Static Analysis Symposium (SAS'99)*, volume 1694 of *LNCS*, pages 39–50, Venice (Italy), September 1999. Springer.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976.

[Kil73] G. Kildall. A unified approach to global program optimization. In *1st Symposium on Principles of Programming Languages (POPL'73)*, pages 194–206, Boston (USA), October 1973. ACM Press, New York.

[KL88] B. Korel and J. W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, November 1988.

[Knu62] D. E. Knuth. *The Art of Computer Programming.* Addison-Wesley, 1962.

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *7th Static Analysis Symposium (SAS'00)*, volume 1824 of *LNCS*, pages 280–301, Santa Barbara (USA), June 2000. Springer.

[LY05] T. Lindholm and F. Yellin. *The Java(tm) Virtual Machine Specification.* SUN, 2005.

[Mau99] L. Mauborgne. *Representation of Sets of Trees for Abstract Interpretation.* PhD thesis, École Polytechnique, 1999.

[Mau00] L. Mauborgne. Tree schemata and fair termination. In *7th Static Analyis Symposium (SAS'00)*, volume 1824 of *LNCS*, pages 302–320, Santa Barbara (USA), June 2000. Springer.

[MCG$^+$99] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, and D. Walker. TALx86: A Realistic Typed Assembly Language. In *1999 ACM SIGPLAN Workshop on Compiler Support for System Software*, pages 25–35, Atlanta (USA), may 1999.

[Mil90] Robin Milner. Operational and algebraic semantics of concurrent processes. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 1201–1242. Elsevier and MIT Press, 1990.

[Min01] A. Miné. The Octagon Abstract Domain. In *Analysis, Slicing and Transformation (in WCRE)*, pages 310–319, Stuttgart (Germany), October 2001. IEEE Computer Society Press.

[Min04a] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *European Symposium On Programming (ESOP'04)*, volume 2986 of *LNCS*, pages 3–17. Springer, April 2004.

[Min04b] A. Miné. *Weakly relational numerical abstract domains.* PhD thesis, École polytechnique, 2004.

[MR03] D. Melski and T. W. Reps. The interprocedural express-lane transformation. In *12th International Conference on Compiler Construction (CC'03)*, volume 2622 of *LNCS*, pages 200–216, Varsaw (Poland), April 2003. Springer.

[MR05] L. Mauborgne and X. Rival. Trace Partitioning in Abstract Interpretation Based Static Analyzers. In *European Symposium On Programming (ESOP'05)*, volume 3444 of *LNCS*, pages 5–20, Edimburgh (UK), April 2005. Springer.

[MT91] R. Milner and M. Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

[MTC$^+$96] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML Compiler: Performance and Safety Through Types. In *1996 ACM SIGPLAN Workshop on Compiler Support for Systems Software*, Tucson (USA), May 1996.

[Nec97] G. C. Necula. Proof-Carrying Code. In *24th Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, January 1997. ACM Press, New York.

[Nec00] G. C. Necula. Translation Validation for an Optimizing Compiler. In *Conference on Programming Language Design and Implementation (PLDI'00)*, pages 83–94, Vancouver, Canada, June 2000. ACM Press, New York.

[NL98] G. C. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Conference on Programming Languages, Design and Implementation (PLDI'98)*, pages 162–173, Montréal, Canada, November 1998. ACM Press.

[NMW02] G. C. Necula, S. McPeak, and W. Weimer. Ccured: type-safe retrofitting of legacy code. In *29th Symposium on Principles of Programming Languages (POPL'02)*, pages 128–139, Portland, Oregon, January 2002. ACM Press, New York.

[OCa] OCaml. The objective caml system. `http://paulliac.inria.fr/ocaml`.

[Par66] R. Parikh. On context-free languages. *Journal of the ACM*, 13(4):570–581, October 1966.

[Pau94] L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *LNCS*. Springer, 1994. with contributions by Tobias Nipkow.

[PHR04] G. J. Pace, N. Halbwachs, and P. Raymond. Counter-example generation in symbolic abstract model-checking. *Software and Tools for Technology Transfer (STTT)*, 5(2-3):158–164, March 2004.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, September 1981.

[PSS98] A. Pnueli, O. Shtrichman, and M. Siegel. Translation Validation for Synchronous Languages. In *25th International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *LNCS*, pages 235–246, Aalborg (Denmark), July 1998. Springer.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *22nd Symposium on Principles of Programming Languages (POPL'95)*, pages 49–61, San Francisco (USA), January 1995. ACM Press, New York.

[Riv03] X. Rival. Abstract Interpretation-based Certification of Assembly Code. In *4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'03)*, volume 2575 of *LNCS*, pages 41–55, New York (USA), January 2003. Springer.

[Riv04a] X. Rival. Invariant Translation-based Certification of Assembly Code. *Software and Tools for Technology Transfer*, 6(1):15–37, July 2004.

[Riv04b] X. Rival. Symbolic transfer functions-based approaches to certified compilation. In *31st Symposium on Principles of Programming Languages (POPL'04)*, pages 1–13, Venice (Italy), January 2004. ACM Press, New York.

[Riv05a] X. Rival. Abstract dependences for alarm diagnosis. In *6th Asian Symposium on Programming Languages and Systems (APLAS'05)*, volume 3780 of *LNCS*, pages 347–363, Tsukuba (Japan), November 2005. Springer.

[Riv05b] X. Rival. Understanding the origin of alarms in ASTRÉE. In *12th Static Analysis Symposium (SAS'05)*, volume 3672 of *LNCS*, pages 303–319, London (UK), September 2005. Springer.

[Sco70] D. Scott. Outline of a mathematical theory of computation. Technical monograph, Oxford University Computing Lab, Programming Research Group, 1970.

[SM03] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security*, 21(1):5–19, 2003.

[SP81] M. Sharir and A. Pnuelli. Two approaches to interprocedural data flow analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–233. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions On Programming Languages And Systems (TOPLAS)*, 24(3):217–298, 2002.

[Str02] M. Strecker. Formal verification of a Java compiler in Isabelle. In *Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 63–77, Copenhagen (Denmark), July 2002. Springer.

[Tar55] A. Tarski. A lattice theoretical fixpoint theorem and its application. *Pacific Journal of Mathematics*, 5:285–310, 1955.

[TCoA99] Radio Technical Commission on Aviation. DO-178B. Technical report, Software Considerations in Airborne Systems and Equipment Certification, 1999.

[TF98] H. Theiling and C. Ferdinand. Combining Abstract Interpretation and ILP for Microarchitecture Modelling and Program Path Analysis. In *19th IEEE Real-Time Systems Symposium*, pages 144–153, Madrid (Spain), December 1998. IEEE Computer Society Press.

[TFW00] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Seperate Cache and Path Analyses. *Real-Time Systems*, 18(2/3), May 2000.

[TG00a] C. Tice and S. L. Graham. Key Instructions: Solving the Code Location Problem for Optimized Code. Research Report 164, Compaq Systems Research Center, september 2000.

[TG00b] C. Tice and S. L. Graham. A Practical, Robust Method for Generating Variable Range Tables. Research Report 165, Compaq Systems Research Center, September 2000.

[TMC$^+$96] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Conference on Programming Language Design and Implementation (PLDI'96)*, pages 181–192, Philadelphia (USA), May 1996. ACM Press.

[VB04] A. Venet and G. Brat. Precise and efficient array bound checking for large embedded C programs. In *Conference on Programming Languages, Design and Implementation (PLDI'04)*, pages 231–242, Washington (USA), June 2004. ACM Press, New York.

[Ven96] A. Venet. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In *3rd Static Analysis Symposium (SAS'96)*, volume 1145 of *LNCS*, Aachen (Germany), September 1996. Springer.

[Wei81] M. Weiser. Program slicing. In *5th International Conference on Software Engineering*, pages 439–449, May 1981.

[Wil95] A. Wiles. Modular elliptic curves and Fermat's last Theorem. *Annals of Mathematics*, 1995.

[WM94] R. Wilhelm and D. Maurer. *Compiler Design.* Springer, 1994.

[XH01] H. Xi and R. Harper. A dependently typed assembly language. In *International Conference on Functional Programming*, pages 169–180, Florence, Italy, September 2001. IEEE Computer Society Press.

[ZPF$^+$02] L. Zuck, A. Pnueli, Y. Fang, B. Goldberg, and Y. Hu. Translation Run-Time Validation of Optimized Code. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

[ZPFG02] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A Translation Validator for Optimizing Compilers. In *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier Science Publishers, 2002.

List of Figures

List of Definitions

List of Theorems and Lemmata

List of Examples

List of Remarks