Proof preservation and program compilation

César Kunz

ED n° 84: Sciences et technologies de l'information et de la communication

N° attribué par la bibliothèque
|__|__|__|__|__|__|__|__|__|__|

T H È S E

pour obtenir le grade de

DOCTEUR DE L'ECOLE NATIONALE SUPERIEURE DES MINES DE PARIS

Spécialité "Informatique temps réel, robotique et automatique"

présentée et soutenue publiquement par
César Darío KUNZ

le 3 Février 2009

PRÉSERVATION DES PREUVES ET TRANSFORMATION DE PROGRAMMES

Directeur de thèse : Gilles Barthe

Jury

Thomas Jensen	Président
Martin Hofmann	Rapporteur
Xavier Leroy	Rapporteur
Peter Müller	Rapporteur
Gilles Barthe	Examinateur
Benjamin Grégoire	Examinateur
Ricardo Peña	Examinateur

Directeur de thèse
Gilles Barthe

Auteur
César Kunz

Préservation des Preuves et Transformation de Programmes

Résumé

Le paradigme du code mobile consiste à la distribution des applications, telles que les applets ou les scripts Web, transferés à travers un réseau non sécurisé comme Internet, qui sont executées dans des systèmes distants, par exemple un ordinateur, un téléphone mobile ou un PDA (Assistant personnel).

Naturellement, cet environnement peut ouvrir la porte au déploiement de programmes malveillants dans des plateformes distantes. Dans certains cas, la mauvaise conduite du code mobile ne constitue pas un risque grave, par exemple lorsque l'intégrité des données affectées par l'exécution n'est pas critique ou lorsque l'architecture d'exécution impose de fortes contraintes sur les capacités d'exécution du code non sécurisé.

Il y a cependant des situations dans lesquelles il est indispensable de vérifier la correction fonctionnelle du code avant son exécution, par exemple lorsque la confidentialité de données critiques comme l'information des cartes de crédit pourrait être en danger, ou lorsque l'environnement d'exécution ne possède pas un mécanisme spécial pour surveiller la consommation excessive des ressources.

George Necula a proposé une technique pour garantir aux utilisateurs la correction du code sans faire confiance aux producteurs. Cette technique, Proof Carrying Code (PCC) [49, 51], consiste à déploier le code avec une preuve formelle de sa correction. La correction est une propriété inhérente du code reçu qui ne peut pas être directement déduite du producteur du code. Naturellement, cela donne un avantage à PCC quant-aux méthodes basées sur la confiance à l'autorité d'un tiers. En effet, une signature d'une autorité ne suffit pas à fournir une confiance absolue sur l'exécution du code reçu.

Depuis les origines du PCC, le type de mécanisme utilisé pour générer des certificats repose sur des analyses statiques qui font partie du compilateur. Par conséquent, en restant automatique, il est intrinsèquement limité à des propriétés très simples. L'augmentation de l'ensemble des propriétés à considerer est difficile et, dans la plupart des cas, exige l'interaction humaine. Une possibilité consiste à vérifier directement le code exécutable. Toutefois, l'absence de structure rend la vérification du code de bas niveau moins naturelle, et donc plus laborieuse. Ceci, combiné avec le fait que la plupart des outils de vérification sont developés pour le code de haut niveau, rend plus appropriée l'idée de transferer la production de certificats au niveau du code source. Le principal inconvénient de produire des certificats pour assurer l'exactitude du code source est que les preuves ne comportent pas la correction du code compilé. Plusieurs techniques peuvent etre proposées pour transférer la preuve de correction d'un programme à sa version exécutable. Cela implique, par

exemple, de déployer le programme source et ses certificats originaux (en plus du code exécutable) et de certifier la correction du processus de compilation. Toutefois, cette approche n'est pas satisfaisante, car en plus d'exiger la disponibilité du code source, la longueur du certificat garantissant la correction du compilation peut être prohibitive.

Une alternative plus viable consiste à proposer un mécanisme permettant de générer des certificats de code compilé à partir des certificats du programme source. Les compilateurs sont des algorithmes complexes composées de plusieurs étapes, parmi lesquelles le programme original est progressivement transformé en représentations intermédiaires. Barthe et al. [14] et Pavlova [57] ont montré que les certificats originaux sont conservés, à quelques différences non significatives près, par la première phase de compilation : la compilation non optimale du code source vers une représentation non structurée de niveau intermédiaire. Toutefois, les optimisations des compilateurs sur les représentations intermédiaires représentent un défi, car a priori les certificats ne peuvent pas être réutilisés. Dans cette thèse, nous analysons comment les optimisations affectent la validité des certificats et nous proposons un mécanisme, *certificate translation*, qui rend possible la génération de certificats pour le code mobile exécutable à partir des certificats au niveau du code source. Cela implique transformer les certificats pour surmonter les effets des optimisations de programme.

Thesis advisor
Gilles Barthe

Author
César Kunz

Certificate Translation alongside Program Transformations

Abstract

Software applications have gained a notable role in our everyday activities, mobile code applications being a significant portion of these software agents. The mobile code paradigm entails the distribution of applications from the code producer to heterogeneous client environments in which they are executed. An extended practice of this paradigm consists in the development of third party components that are transferred across an untrusted network such as the Internet and finally integrated in a host execution environment such as a PC or a cellular phone.

Naturally, this computational environment opens the door to the deployment of malicious code in client workstations. In some cases a potential misbehavior of the mobile code does not constitute a serious risk, for example when the integrity of the data affected by the execution is irrelevant, or when the execution architecture imposes strong constraints on the computational capabilities of the foreign application.

There still are, however, situations in which it is essential to verify the functional correctness of the code before executing it, for instance when the confidentiality of critical data such as credit card information could be compromised, or when the execution environment does not have a special mechanism to monitor excessive resource consumption.

George Necula proposed a technique to provide trust to code consumers about the program correctness without trusting the code producer side. The technique, Proof Carrying Code (PCC) [49, 51], consists in deploying code together with a formal proof of its correctness. Correctness is an inherent property of the received code that cannot be inferred from the identity of the code producer. That naturally puts PCC in advantage with respect to methods based on trusting a third party authority. Indeed, a signature from a trusted authority it is not sufficient to provide absolute trust on the execution of the received code.

From its origins, the typical mechanism of PCC to generate certificates relies on a certifying compiler, an extension of a standard compiler that automatically produces certificates for decidable safety policies. Extending the set of enforceable properties is challenging and in the most general case it requires human interaction. One possibility is to verify the final executable code from scratch. However, the lack of structure makes low level code verification a less natural and, hence, more daunting task. This, together with the fact that verification environments target mostly high level code, makes more appropriate the idea of moving the generation of certificates at the source code level. The main drawback of producing certificates ensuring source code correctness is that they do not entail correctness of the final compiled code.

Several techniques may be proposed to transfer evidence of program correctness from the source code to the executable counterpart. That includes, for instance, deploying the source program and original certificates in addition to the executable code and certifying the correctness of the compilation process. However, this approach is not satisfactory, since in addition to require availability of the source code, a certificate ensuring compiler correctness can be prohibitively large.

A more viable alternative is to provide a mechanism to generate certificates for compiled code from certificates of the original source program. Compilers are complex procedures composed of several steps, in which the original program is progressively transformed into intermediate lower level representations. Barthe et al. [14] and Pavlova [57] have shown that original certificates are preserved, up to minor differences, along with the first compiler phase: a nonoptimizing compilation from source level to an unstructured intermediate representation. However, compiler optimizations applied to the intermediate representations represent a challenge since a priori certificates cannot be reused. In this thesis, we analyze how optimizations affect the validity of certificates and propose a mechanism, Certificate Translation, to transform certificates in order to overcome the effects of program transformations, rendering feasible the generation of certificates for executable mobile code at the source level.

Chapter 2 introduces certificate translation for common compiler optimizations applied in the context of an intermediate RTL language. The main difficulties are presented, and a classification of program transformations is given in terms of the effort required to transform the original certificates. A general scheme is provided to cover transformations that perform arithmetic simplifications such as constant propagation and common subexpression elimination. In addition, ad-hoc techniques are proposed for other standard optimizations that do not correspond to this classification. Chapter 3 studies the existence of certificate translators in a mild extension of an abstract interpretation framework that includes a notion of certificates. This abstract setting provides considerable advantages with respect to the approach of Chapter 2 since it allows us to extend certificate translation to diverse programming languages and several analysis environments. Certificate transformation is studied in this abstract setting for simple program transformations which can be composed to represent a wide variety of program optimizations, for instance those considered in Chapter 2. In a second part of the thesis, we extend certificate translation to less typical settings, which can be of interest in PCC scenarios. More precisely, the chapters in the second part consider the aspect oriented paradigm, hybrid verification methods, and parallel programs executing in memory hierarchies. For each setting, appropriate analysis and verification settings are provided. Then, the effect of common transformations over verification results are studied.

Acknowledgments

First, I would like to express my gratitude to my supervisor Gilles Barthe for all the scientific support and for giving me the chance to pursuit this project, and also to Daniel Fridlender for motivating me to continue my doctoral studies at INRIA in France. To Benjamin Gregoire, for his technical discussions and demanding criticisms, and for the enthusiasm he showed and spread to me right after the first results on certificate translation. To the dissertation reviewers, Xavier Leroy, Peter Müller and Martin Hoffman for the time and effort spend when reading and criticizing the thesis, to Thomas Jensen for accepting being president of the jury, and the rest of the committee for participating on the thesis defense. It's been an honor to count on them.

I would like to especially thank all the people that has made my PhD studies an enjoyable period of time. To Tamara Rezk, for helping me to take my first steps in my research career and, more importantly, for her supporting friendship along the road to my PhD title. To Jimena Costa, whose presence at INRIA was really valuable to help me integrate myself into a foreign environment, and to Dante Zanarini and Jorge Sacchini, who have shared with me the first days in France. To the rest of my friends with whom I shared a lot of memorable moments: Gustavo Petri, Santiago Zanella, Mariela Pavlova, Fernando Pastawski, Andres Krapf; to Gabriela Gallegos, who let me enjoy her mexican traditions, my chilean friends, and the rest of the members of the INRIA Everest and Marelle teams. To Barbara André, for her especially enriching company during my last year in France, and all my other french friends that forced me to practice their native tongue. To Nathalie Bellesso, the Everest team project assistant, not only for her efficiency but for her warm kindness that made me feel welcome in the project. To the IMDEA Software Institute, for the financial support in the last months before the thesis defense, but specially for the help received from María Alcaraz and Paola Huerta, which I really appreciate.

My doctoral research activities has been funded by the European project MOBIUS (FP6-015905).

to my little niece ***Milagros****, who suffered my absence from her very early age.*

Contents

1

Introduction

The mobile code paradigm entails the distribution of applications from the code producer to heterogeneous client environments in which they are executed. An extended practice of this paradigm consists in the development of third party components, which are transferred across an untrusted network such as the Internet and finally integrated in a host execution environment such as a PC or a cellular phone. Naturally, this computational environment opens the door to the deployment of malicious code in client workstations. In some cases a potential misbehavior of the mobile code does not constitute a serious risk, for example when the integrity of the data affected by the execution is irrelevant or when the execution architecture imposes strong constraints on the computational capabilities of the foreign application. In other cases, it is sufficient to automatically estimate a bound of the resource consumption or check whether confidential data is not leaked. There still are, however, situations in which it is essential to verify more complex properties. For instance when a downloaded component is required to follow a particular functional specification in order to match a particular API, or when the program complexity renders hard to verify a simple confidentiality property.

George Necula proposed a technique to provide trust to code consumers about the program correctness, dispensing them from trusting code producers (that are potentially malicious), networks (that may be controlled by an attacker), and compilers (that may be buggy). The technique, Proof Carrying Code (PCC) [49, 51], consists in requiring mobile code to be sent along with verifiable evidence of their adherence to an appropriate policy that may involve requirements about their safety, security, or functionality.

PCC benefits from a number of advantageous features. First, PCC is based on verification rather than trust. Indeed, PCC focuses on correctness, which is an inherent property of the received code that cannot be directly inferred from trust in the code producer. Second, it is transparent for end users, since they are not required to build proofs. Rather, it requires code consumers to check proofs, which is fully automatic. Third, the principle of Proof Carrying Code is general; the only restriction on the security policy is that it should

be expressible in the formal logic, which is often very expressive. Finally, PCC technology does not sacrifice performance to security as it advocates for static verification at compile-time, and therefore does not incur in the overhead cost inherent to dynamic techniques based on monitoring.

A PCC infrastructure builds upon several elements: a logic, a verification condition generator, a formal representation of proofs, and a proof checker. Figure 1.1 shows a scheme of the client side of a PCC architecture. We briefly

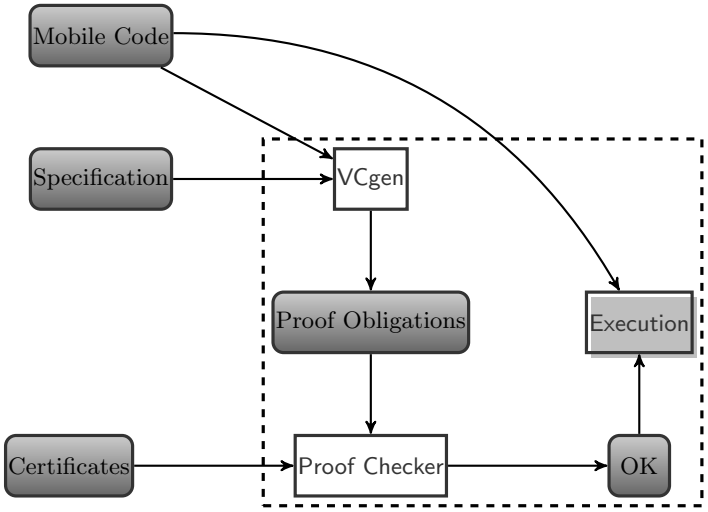


Fig. 1.1. PCC Scheme - Client Side.

describe each component:

A formal logic for specifying and verifying policies. The specification is used to express the expected requirements on the incoming component. PCC adopts first-order predicate logic as a formalism to both specify and verify the correctness of components, and focuses on safety properties. Thus, requirements are expressed as a pre and postcondition relating the initial and final states of a function invocation.

A verification condition generator (VCgen). For each component, the VCgen produces a set of proof obligations whose validity will be sufficient to ensure that the component respects the safety policy. PCC adopts a VC-gen based on programming verification techniques such as Hoare-Floyd logics and weakest precondition calculi, and it requires that components come equipped with extra annotations, e.g., loop invariants that make the generation of verification conditions feasible.

Certificates. A certificate is a formal representation of proofs that conveys easy-to-verify evidence of the validity of proof obligations and, thus, that

the code is correct. Commonly, certificates are terms of the lambda calculus, as suggested by the Curry-Howard isomorphism, and routinely used in modern proof assistants such as Coq and LF.

A proof checker. The objective of a proof checker is to verify that the certificate does indeed establish the proof obligations generated by the VC Generator. Proof checking can be reduced to type checking by virtue of the Curry-Howard isomorphism, so that the proof checker only needs to verify that the certificate is of the correct type. One very attractive advantage of this approach is the simplicity of the proof checker, which forms part of the client environment.

The Trusted Computing Base (TCB) of a PCC environment is the set of components that must be trusted in order to ensure the soundness of a certified program. Any bug in the components outside the TCB does not affect the soundness of the code execution. For instance, in the overall scheme shown in Figure 1.1, it is essential and sufficient to rely on the correctness of the VCgen, the Proof Checker, and the execution environment.

Figure 1.2 shows an overall representation of a typical automatic certificate generation. Certifying compilers [51] extend traditional compilers with a mechanism to automatically generate certificates for sufficiently simple safety properties, exploiting the information available about a program during its compilation. Note that the certifying compiler does not form part of the client TCB; nevertheless, it is an essential ingredient of PCC, since it reduces the burden of verification on the code producer side.

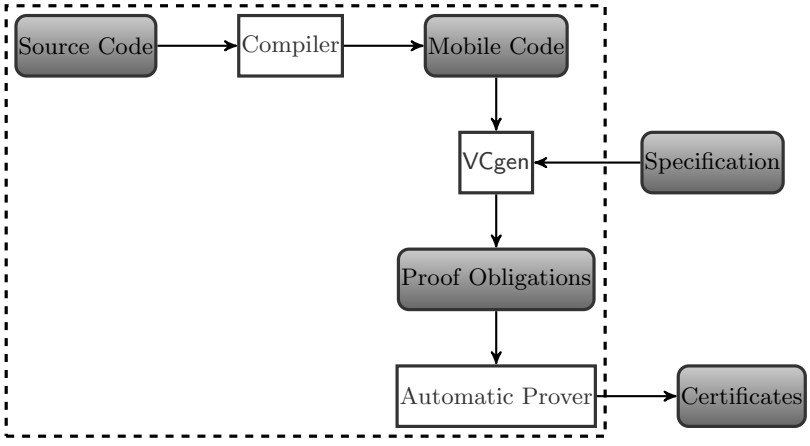


Fig. 1.2. Code Producer Side - Typical Certificate Generation

While certificate checking is reasonably understood, certificate generation remains a challenging problem.

An early example of certifying compiler is the Touchstone compiler [51], which was intended to explore the feasibility of PCC. This compiler generates, for programs written in a type-safe fragment of C, a formal proof for type-based safety and memory safety of the resulting program in DEC Alpha assembly language. The Touchstone compiler automatically inserts the loops invariants in the resulting program and generates the correctness proofs. Certifying compilation is by design restricted to a specific class of properties and programs— in order to achieve automatic generation of certificates and, thus, to reduce the burden of verification on the code producer side. The counterpart of this approach is that the ensured properties are restricted to simple properties, namely typing predicates.

While it is possible to generate certificates automatically for properties that are enforceable by automated program analyses, and in particular type systems, certificate generation remains necessarily interactive in the general case. For instance, in situations where the functional correctness of the downloaded code is essential, and where certificate issues such as size or checking time are not relevant.

In order to enforce arbitrary properties on arbitrary programs, there are several frameworks to verify low-level code but little work studying the link between reasoning at source and compiled levels. The essence of certificate translation is to extend standard compilers to automatically transfer the result of formal source code verification to a certificate for the compiled code.

Source code verification is increasingly being used to validate safety-critical or security-critical software [4, 25, 22, 15]. While automated theorem provers are useful to detect many common programming mistakes and sometimes to establish some simple policies, the use of interactive verification tools might be required for many policies, including basic safety policies for complex software (the complexity of the software may render relatively simple safety policies difficult to verify automatically), and complex policies that involve the functional behavior of software. Several program verification environments have been developed for popular programming languages such as Eiffel, Java or C [46, 31]. The focus on high level languages is beneficial for developers, since it enables them to gain increased confidence or feedback directly on their code, without the need to consider the particular runtime environment where the code shall be executed.

The primary objective of *certificate translation* is precisely to overcome the limitation of certifying compilers and extend the scope of PCC to complex policies. To this end, certificate translation transforms certificates of source-language programs into certificates of compiled programs and, thus, fills the gap between widely used interactive source code verification environments and PCC. An overall scheme of certificate generation by proof transformation is shown in Figure 1.3. In the figure, the code producer extracts a set of proof obligations from the source code and the policy specification. Then, certificates entailing source code correctness are generated. Finally, certificates for the

mobile code generated by compilation of the source code are obtained by certificate translation from the certificates of the source program.

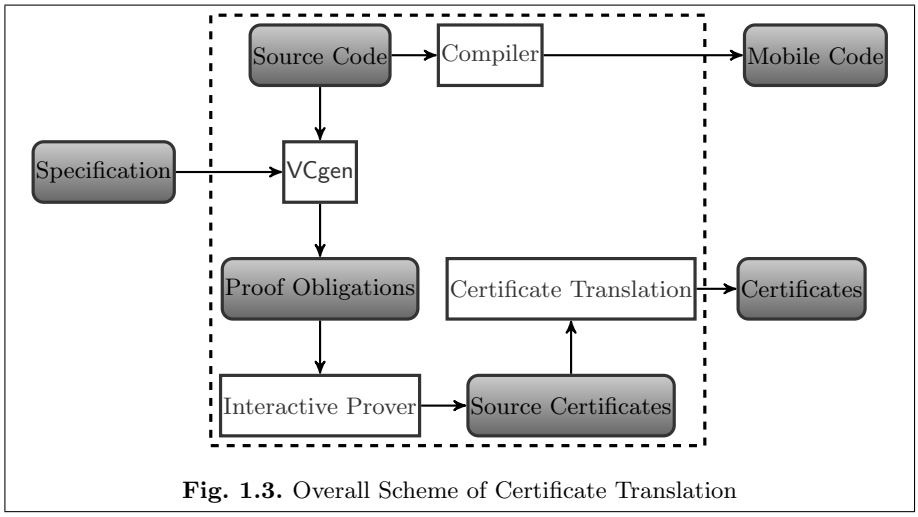


Fig. 1.3. Overall Scheme of Certificate Translation

Given a compiler represented by the function $[\![\cdot]\!]$, a function $[\![\cdot]\!]_{\text{spec}}$ to transform specifications, and certificate checkers (expressed as a ternary relation "c is a certificate that P adheres to ϕ" and written $c : P \models \phi$), a certificate translator is a function $[\![\cdot]\!]_{\text{cert}}$ such that for all programs p, policies ϕ, and certificates c,

$$\text{if} \quad c : p \models \phi \quad \text{then} \quad [\![c]\!]_{\text{cert}} : [\![p]\!] \models [\![\phi]\!]_{\text{spec}}$$

Building a certificate translator for nonoptimizing compilation is relatively simple since proof obligations are preserved (up to minor differences), and hence it is possible to reuse the certificates of source code programs for the compilation result; see e.g. the work of Barthe et al. [14] and Pavlova [57]. However, program optimizations successively applied at the intermediate compiler phases do not preserve proof obligations and, hence, certificates cannot be reused. It is precisely the object of this thesis to study how to overcome the proof transformations that are needed to translate certificates in the presence of program optimizations. For each optimization step, we show how to appropriately transform the certificates. Then, they can be combined to translate certificates for the complete compilation process.

Outline and Sources

Part I: Foundations of Certificate Translation

The first part focuses on the existence of certificate translation procedures for standard compiler optimizations.

Chapter 2: In Chapter 2, we study the existence of certificate translation procedures for standard program optimizations applied at intermediate compilation stages.

As a verification environment, we adopt a weakest-precondition-based VC-gen. Program verification consists in extracting from a partially annotated program a set of verification conditions, which must be discharged in order to prove the program correct. To represent the verification condition proofs that are subject to transformation, we propose an abstract notion of proof algebra. Then, certificate translators are defined from a set of functions, closed in the domain of certificates, which represent the application of logic rules, such as introduction and elimination for the $\wedge$ and $\vee$ connective and for the $\forall$ quantifier.

As stated in common compiler textbooks, compilation proceeds by successive transformations that gradually translate the code into a series of lower level representations, bringing the program closer to the actual executable encoding. For each step, we build an appropriate certificate translator, paying special attention on standard optimizations applied to an intermediate Register Transfer Language (RTL) representation. The set of optimizations includes Constant Propagation, Loop Induction Variable Strength Reduction, Common Subexpression Elimination, Copy Propagation, Dead Register Elimination, Unreachable Code Elimination, Register Allocation, and Function Inlining.

Commonly, compiler optimizations are performed in two phases: first, a static analyzer computes some information from the syntactic program representation and then, on the basis of this information, a semantics-preserving transformation is applied. For some optimizations that perform arithmetic simplifications, the existence of a certificate translation procedure requires a formal proof ensuring the correctness of the result of the analysis. We show that, for the standard optimizations considered in this chapter, it is feasible to define an automatic procedure to construct a certificate for the logical interpretation of the result of the analysis. We leave for Chapter 3 a formalization of the set of requirements in order to define a certifying analyzer. For other optimizations that do not fit in this category, e.g. dead variable elimination, an ad-hoc certificate translator is proposed.

This work is an extension of the refereed paper **Certificate Translation for Optimizing Compilers** [7], coauthored with Gilles Barthe, Benjamin Grégoire and Tamara Rezk, presented at the 13^{th} International Static Analysis Symposium (SAS 2006). A large portion of this chapter has been accepted for publication in the ACM Transactions on Programming Languages and Systems (TOPLAS).

Chapter 3: In this chapter, we take a more general approach and formalize the results of Chapter 2 under an abstract interpretation framework, which is a unifying model for the two main components of a certificate translation procedure: the verification environment in which the original certificate

is produced and the static analysis that justifies a program optimization. The choice of an abstract framework provides a substantial leverage with respect to previous results, since it enables us to extend our results to a wider set of programming languages and verification environments.
In the abstract setting, annotations are represented as partial labellings from program nodes to abstract values and verification consists in checking whether the labeling satisfies a set of inequalities in the abstract domain. To facilitate checking that the labeling bound to the program satisfies the required constraints, we provide a mild extension of the abstract interpretation model to incorporate a certificate infrastructure. One can see that the VCgen framework defined in Chapter 2 is a particular instance of the extended abstract interpretation framework.
As stated above, a certifying analyzer is a main component in the definition of a certificate translator. We show in this chapter that proving the existence of a certifying analyzer boils down to discharging the verification conditions required to prove consistency of the analysis with respect to the verification environment.
Instead of considering specific program optimizations, we study certificate translation in the presence of basic transformations that, combined, can represent a wide range of program optimizations such as those presented in Chapter 2.
The contents of this chapter are an extension of the refereed paper **Certificate Translation in Abstract Interpretation** [10] coauthored with Gilles Barthe, presented at the 17$^{\text{th}}$ European Symposium on Programming (ESOP 2008).

Part II: Case Studies

In this part, we extend the application of certificate translation to less typical programming and verification scenarios.

Chapter 4: Aspect Oriented Programming (AOP) has an interesting application in the development of Proof Carrying Code scenarios, since it enables one to separate concerns such as resource control and security from the basic functionality of an application. In this incremental development process, the expected functionality can be provided by a baseline program, and successive refinements that improve non-functional concerns can be possibly provided by third parties.
In Chapter 4, we explore a PCC architecture that accommodates an incremental development process, based on aspect oriented programming, and extend the results of Part I to show, in the context of a Simple AOP Language (SAL), that it is possible to generate certificates of executable code from proofs of aspect oriented programs. To achieve this goal, we introduce a notion of specification preserving advice, and provide a verification method for programs with specification-preserving advice. Informally, an advice a is specification-preserving for an annotated piece of

code $\{\Phi\}c\{\Psi\}$, where Φ and Ψ denote the pre and postcondition for c, respectively, if the result of weaving the advise a with program c satisfies the same pre and postcondition.

A specification-preserving advice is natural in the context of PCC with intermediaries, since many aspects related to security (resource management, logging, *etc.*) and efficiency (e.g., cached functions, optimized code, *etc.*) fall in this category. Another advantage is that a specification-preserving advice support "separate verification", allowing intermediaries to treat correctness proofs of the baseline code and the advice as black-boxes.

We define a simple stack-based language (SBL) into which the simple AOP programs can be compiled, and a VCgen for SBL programs. We then show that correct SAL programs are compiled into correct SBL programs.

Chapter 4 is based on the refereed paper **Certificate translation for specification-preserving advice** [9], coauthored with Gilles Barthe, presented at the Foundations of Aspect-Oriented Languages workshop (FOAL 2008).

Chapter 5: In order to reduce the verification effort, hybrid verification environments increasingly rely on combining static analyses and verification condition generation. The VCgen exploits the information of the analysis in two useful ways: on the one hand, verification conditions that originate from spurious edges in the control-flow graph are discarded. This leads to fewer and smaller proof obligations. Furthermore, the VCgen adds the results of the analysis as additional assumptions to help prove the verification conditions.

In Chapter 5, we introduce two hybrid verification frameworks for source code and low level bytecode, respectively. The objective of this work is to extend preservation of proof obligations to hybrid verification methods. One well-known difficulty with the preservation of static analysis results is the loss of precision incurred by compilation [45]. To solve this difficulty, we achieve preservation of solutions by defining at bytecode level a symbolic execution that decompiles stack instructions. Finally, we show that programs that are provably correct using the hybrid method are provably correct using standard VCgen. To this end, we define a compiler that transforms a hybrid specification (merging logical assertions and analysis results) into a standard one by giving a logical interpretation of the analysis results. This result ensures soundness of the hybrid verification method from the soundness of the standard VCgen.

The contents of Chapter 5 are presented in the refereed paper **Preservation of proof obligations for hybrid verification methods** [11], coauthored with Gilles Barthe, David Pichardie, and Julian Samborski-Forlese, published at the 6$^{\text{th}}$ IEEE International Conference on Software Engineering and Formal Methods (SEFM 2008).

Chapter 6: As parallel programming languages for high-performance computing are increasingly gaining wide acceptance, there is a need to provide

analysis and verification methods to help developers write, maintain, and optimize their high-performance applications.

In Chapter 6, we propose a framework to specify and certify the behavior of parallel divide-and-conquer programs over hierarchical memories. Sequoia is a special purpose program representation to exploit the high-performance support offered by modern computer architectures. The language contains explicit constructions to successively fragment computations into smaller tasks, which are executed down in the memory hierarchy. We use the framework of abstract interpretation to design analysis and verification frameworks to reason about Sequoia programs. A main component of this framework is an analysis that checks whether subtasks operate over disjoint portions of the memory. In this setting, we study whether a certificate translator procedure can be defined for optimizations that are typical for explicitly parallel tasks executing in hierarchical memories.

Chapter 6 is an extension of the paper **Certified Reasoning in Memory Hierarchies** [12], coauthored with Gilles Barthe and Jorge Sacchini, published at the 6^{th} ASIAN Symposium on Programming Languages and Systems (APLAS 2008).

Part I

Foundations of Certificate Translation

2

Certificate Translation alongside Standard RTL Optimizations

This chapter outlines the principles of certificate translation, and instantiates it in the context of an optimizing compiler from a high-level imperative language to an intermediate RTL representation. Compilation proceeds by successive transformations: imperative programs are first translated into RTL programs, then common program optimizations are successively applied to RTL programs in order to produce the final RTL program. For completeness, we first state a well-known result for the first compilation phase [14, 8]: proof obligations are preserved, up to minor differences, by a nonoptimizing compiler from the high-level to RTL representation. Then, for each of the succesive optimizations applied to the RTL program representation, we define an appropriate certificate translator.

The verification infrastructure builds upon verification condition generators (VCgen), which are part of the standard PCC infrastructure and are also used in many interactive verification environments. A VCgen can be seen as an automatic strategy for applying Hoare logic rules; it generates from a program p and a specification ϕ a set of proof obligations $\mathsf{PO}(p, \phi)$ whose validity ensures that the program meets its specification. In this setting, c is a certificate that p satisfies ϕ (denoted $c : p \models \phi$) iff c is a set of (logical) certificates such that for every proof obligation $\psi \in \mathsf{PO}(p, \phi)$, there is a (logical) certificate $d \in c$ whose mere existence ensures the validity of ψ, denoted with the binary relation $d \models_{\mathrm{po}} \psi$.

Of course, certificate translation also depends by definition on the format of certificates, and on the procedure to check that a certificate establishes a property ψ. Nevertheless, one does not need to commit to a particular certificate infrastructure for proof obligations in order to study the existence of certificate translators. Instead, the existence of certificate translators for common program optimizations is shown under the assumption that certificates are closed under a few logical rules that include introduction and elimination rules for the $\wedge$ and $\Rightarrow$ connectives and for the $\forall$ quantifier, as well as substitution of the subexpressions e_1 by e_2 (or viceversa) under the hypothesis $e_1 = e_2$.

Difficulties

Building a certificate translator for non-optimizing compilation is relatively simple since proof obligations are preserved up to minor differences. Dealing with optimizations is more challenging because:

- *Optimizations that perform arithmetic simplifications* such as constant propagation or common subexpression elimination, do not necessarily preserve verification conditions. Consider the following piece of code to which constant propagation is applied:

$$\begin{array}{ll} r_1 := 1 & r_1 := 1 \\ \{\mathsf{true}\} & \{\mathsf{true}\} \\ r_2 := r_1 & r_2 := 1 \\ \{r_1 = r_2\} & \{r_1 = r_2\} \end{array}$$

  Proof obligations for the assignment instruction to r_2 are $\mathsf{true} \Rightarrow r_1 = r_1$ and $\mathsf{true} \Rightarrow r_1 = 1$ for the original and optimized version, respectively. The second proof obligation is unprovable, since it is unrelated to the sequence of code containing the assignment $r_1 := 1$.
  Remark: Notice that, for the short code fragment above, proof obligations do not coincide due to our particular definition of wp. Consider a more complex $VCgen$ that propagates the computation of wp for the assignment $r_1 := 1$ over the proof obligations $\mathsf{true} \Rightarrow r_1 = r_1$ and $\mathsf{true} \Rightarrow r_1 = 1$. Such alternative $VCgen$ returns proof obligations that are identical when given the example above as input. Notice, however, there are still cases in which a more complex $VCgen$ also fails, for instance, when given as input an alternative example

$$\begin{array}{l} c \\ \{\mathsf{true}\} \\ r_2 := r_1 \\ \{r_1 = r_2\} \end{array}$$

  where c is a non-trivial loop from which the analysis can infer the postcondition $r_1 = 1$. *End of Remark.*

  The conditions that justify an optimization opportunity must be propagated through every intermediate assertion. Therefore, typical analyzers must be extended into *certifying analyzers*, which justify analyses upon which the optimizations rely by expressing their results in the logic of the PCC architecture, and produce a certificate of the analysis for each program. Then, a function weaves the certificate of the original program and the certificate produced by the certifying analyzer, to produce the certificate of the optimized program. The process is presented in Figure 2.1;
- *Optimizations that eliminate instructions* without computational role (assignments to dead registers, nop instructions) may also eliminate information that is required to prove the program correct. For example, eliminating

nop instructions may lead to delete assertions attached to them, or dead register elimination may remove registers that occur in intermediate assertions of the program. Considering the following transformation:

$$\begin{array}{l} x := n; \\ \vdots \\ \{x = n\} \\ y := x; \\ \vdots \\ \{x = y\} \end{array} \longrightarrow \begin{array}{l} x := n; \\ \vdots \\ \{x = n\} \\ y := n; \\ \vdots \\ \{x = y\} \end{array}$$

After performing constant propagation, the variable x becomes dead. However, one cannot simply remove the first assignment since proof obligations referring to dead registers ($\{x = n\}$ in this case) cannot be proved because all hypotheses about these registers would be lost. Thus, in order to define a certificate translator for dead register elimination, we propose a different kind of transformation that performs simultaneously dead variable elimination in instructions and in assertions.

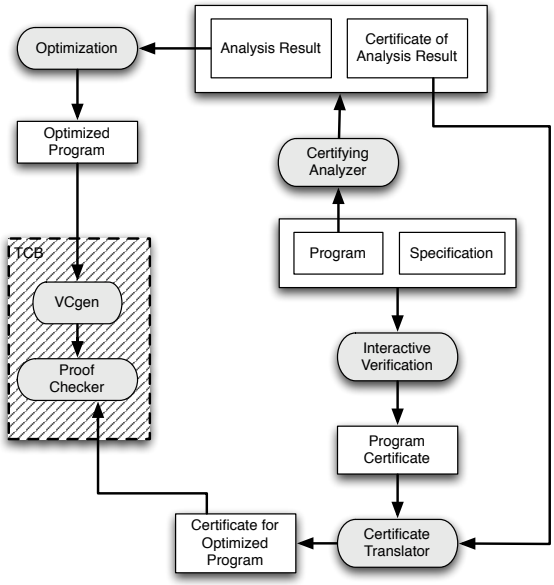


Fig. 2.1. Overall picture of certificate translation

According to the characteristics of their certificate translators, optimizations fall in one of the following categories:

— PPO (Preservation of Proof Obligations): PPO deals with transformations for which the annotations are not rewritten, and where the proof obligations (for the original and transformed programs) coincide. This category covers transformations such as nonoptimizing compilation and unreachable code elimination;
— IPO (Instantiation of Proof Obligations): IPO deals with transformations where the annotations and proof obligations for the transformed program can be described as a variable renaming of the annotations and proof obligations for the original program, thus certificates are translated by α-equivalence. This category covers dead register elimination and register allocation;
— SCT (Standard Certificate Translation): SCT deals with transformations for which the annotations are not rewritten, but where the verification conditions do not coincide. This category covers transformations such as loop unrolling and function inlining;
— CTCA (Certificate Translation with Certifying Analyzers): CTCA deals with transformations for which the annotations need to be rewritten, and for which certificate translation relies on having certified previously the analysis used by the transformation, using *certifying analyzers* that produce a certificate that the analyzer is correct on the source program. This category covers constant propagation, common subexpression elimination, strength reduction, and other optimizations that rely on arithmetic.

2.0.1 Contents

This section:

- introduces certificate translation as a means to extend significantly the scope of PCC to complex security policies;
- classifies certificate translation for common optimizations, including constant propagation, loop induction variable strength reduction, dead register elimination, common subexpression elimination, copy propagation, unreachable code elimination, register allocation, and function inlining. We present each of the certificate translators for the RTL language.

In Section 2.1, we introduce our proof carrying code setting, including a Register Transfer Language (RTL) and its verification infrastructure. In Section 2.2, we present certificate translation for nonoptimizing compilers. In Section 2.3, we describe certificate translation for several standard optimizations.

2.1 PCC Setting

2.1.1 RTL Language

Our language RTL (Register Transfer Language) is a low-level, side-effect free, language with conditional jumps and function calls, extended with annotations drawn from a suitable assertion language. The choice of the assertion language does not affect our results, provided assertions are closed under the connectives and operations that are used by the verification condition generator.

The syntax of expressions, formulas, and RTL programs (suitably extended to accommodate certificates, see Section 2.1.4), is shown in Figure 2.2, where $n \in \mathbb{N}$ and $r \in \mathcal{R}$, with $\mathcal{R}$ an infinite set of register names. We let φ, ϕ, and ψ range over assertions. A program p is defined as a function from RTL func-

comparison	$\lhd$	$::= < \mid \leq \mid = \mid \geq \mid >$
expressions	e	$::= n \mid r \mid -e \mid e + e \mid e * e \mid \ldots$
assertions	φ	$::= \mathsf{true} \mid e \lhd e \mid \varphi \wedge \varphi \mid \neg\varphi \mid \forall r.\ \varphi \mid \ldots$
comparisons	cmp	$::= r \lhd r \mid r \lhd n$
operators	op	$::= n \mid r \mid n + r \mid \ldots$
instr. desc.	ins	$::= r_d := \mathsf{op},\ L$
		$\mid r_d := f(\vec{r}),\ L$
		$\mid \mathsf{cmp}\ ?\ L_t : L_f$
		$\mid \mathsf{return}\ r$
		$\mid \mathsf{nop},\ L$
instructions	I	$::= (\phi,\ \mathsf{ins}) \mid \mathsf{ins}$
fun. decl	F	$::= \{\vec{r};\ \varphi;\ G;\ \psi;\ \lambda;\ \vec{\Lambda}\}$

Fig. 2.2. Syntax of RTL

tion identifiers to function declarations. RTL functions return integer values. Every program comes equipped with a special function, namely main, and its declaration. The body of a function is defined as a mapping G from program labels to instructions, and instructions are equipped with explicit successors. Hence, the body of a function constitutes a (closed) directed graph. A mapping from program points to instructions is preferred, rather than an instruction sequence, to abstract from the details of label updating when modifying the function code. In the sequel, the distinguished label L_{sp} is used as an entry point for every function body. A declaration F for a function f includes its formal parameters $\vec{r}$, a precondition φ, a (closed) graph code G, a postcondition ψ, a certificate λ, and a function $\vec{\Lambda}$ from reachable labels to certificates (the notion of certificate is defined in Section 2.1.4). For clarity, we often use

a subscript f for referring to elements in the declaration of a function f, e.g., the graph code of a function f as G_f.

As will be defined below, the VCgen generates one proof obligation for each program point containing an annotation plus one proof obligation for the entry point L_{sp}. The component λ is a certificate that attests the validity of the latter proof obligation and $\vec{\Lambda}$ maps every program point that contains an assertion to the certificate of its associated proof obligation.

Formal parameters are represented as a list of registers, from the set $\mathcal{R}$, which we suppose to be local to f. For specification purposes, we introduce for each register r in $\vec{r}$ a (pseudo)register r^*, not appearing in the code of the function, and which represents the initial value of a register declared as formal parameter. We let $\mathcal{R}^*$ denote the set $\{r^* \mid r \in \mathcal{R}\}$ and $\vec{r^*}$ denote a sequence of registers in $\mathcal{R}^*$. We also introduce, for specification purposes, a (pseudo)register res, not appearing in the code of the function, and which represents the result or return value of the function. The annotations φ and ψ in Figure 2.2 provide respectively the specification of pre and postcondition of the function, and are subject to well-formedness constraints. The precondition of a function f, also referred as $\mathsf{pre}(f)$, is an assertion in which the only registers to occur are the function formal parameters, hereafter denoted $\vec{r_f}$; in other words, the precondition of a function can only talk about the initial values of its parameters. The postcondition of a function f, also referred as $\mathsf{post}(f)$, is an assertion[1] in which the only registers to occur are res and the formal parameters $\vec{r_f}^*$; in other words, the postcondition of a function can only talk about its result and the initial values of its parameters.

A graph code of a function is a partial function from labels to instructions. We assume that every graph code includes a special label, namely L_{sp}, corresponding to the starting label of the function, i.e., the first instruction to be executed when the method is called. Given a function f and a label L in the domain of its graph code, we will often use $f[L]$ instead of $G_f(L)$, i.e., the application of graph code of f to label L.

An instruction is either an instruction descriptor ins or a pair $(\phi,\ \mathsf{ins})$ consisting of an annotation ϕ and an instruction descriptor ins. An instruction descriptor can be an assignment, a function call, a conditional jump, or a return instruction. Operations on registers are those of standard processors, such as movement of registers or values into registers $r_d := r$, and arithmetic operations between registers or between a register and a value. Furthermore, every instruction descriptor carries explicitly its successor label(s); due to this mechanism, we do not need to include unconditional jumps, i.e., "goto" instructions, in the language. Immediate successors of a label L in the graph of a function f are denoted by the set $\mathsf{succ}_f(L)$. We assume that the graph is closed; and in particular, if L is associated with a return instruction, $\mathsf{succ}_f(L) = \emptyset$.

[1] Notice that a postcondition is not exactly an assertion in the sense that it uses register names from $\vec{r^*}$, which do not appear in preconditions.

2.1.2 Operational Semantics

The operational semantics of RTL is standard. In particular, neither proofs nor assertions interfere with the semantics. The semantics is defined in Figure 2.3 as a big step relation between non terminal states and a terminal state. A non terminal state is defined as a tuple with two elements: the current instruction and a map ρ from local registers to values. The expression $[\rho \mid x \mapsto n]$ stands for the function ρ' s.t. $\rho' y = n$ if $x = y$ and $\rho' y = \rho y$ otherwise.

$$\frac{\langle \mathsf{ins}, \rho\rangle \rightsquigarrow_f n}{\langle(\varphi,\ \mathsf{ins}), \rho\rangle \rightsquigarrow_f n}$$

$$\frac{\langle f[L], [\rho \mid r_d \mapsto [\![\mathsf{op}]\!]_\rho]\rangle \rightsquigarrow_f n}{\langle r_d := \mathsf{op},\ L, \rho\rangle \rightsquigarrow_f n}$$

$$\frac{\langle f[L_t], \rho\rangle \rightsquigarrow_f n}{\langle \mathsf{cmp}\ ?\ L_t : L_f, \rho\rangle \rightsquigarrow_f n} \text{ if } [\![\langle \mathsf{cmp}\rangle]\!]_\rho$$

$$\frac{\langle f[L_f], \rho\rangle \rightsquigarrow_f n}{\langle \mathsf{cmp}\ ?\ L_t : L_f, \rho\rangle \rightsquigarrow_f n} \text{ if } \neg[\![\langle \mathsf{cmp}\rangle]\!]_\rho$$

$$\frac{\langle g[L_{\mathsf{sp}}], [\vec{r}_g \mapsto \rho\vec{r}]\rangle \rightsquigarrow_g m \quad \langle f[L'], [\rho \mid ret \mapsto m]\rangle \rightsquigarrow_f n}{\langle ret := g(\vec{r}),\ L', \rho\rangle \rightsquigarrow_f n}$$

$$\frac{}{\langle \mathsf{return}\ r, \rho\rangle \rightsquigarrow_f \rho r}$$

Fig. 2.3. Operational Semantics

Let $[\![]\!]$ be a standard interpretation function that takes an assertion and a map from registers to values and returns a logical proposition. For clarity, the interpretation $[\![.]\!]_\rho^{\rho^*}$ refers to two parameters ρ and ρ^* with disjoint domains $\mathcal{R}$ and $\mathcal{R}^*$, respectively. When it is clear from the context that an assertion ϕ does not contain registers in $\mathcal{R}^*$, e.g., when ϕ is a precondition, we may simply write $[\![\phi]\!]_\rho$ instead of $[\![\phi]\!]_\rho^{\rho^*}$.

We say that an assertion is valid, if for every assignments ρ and ρ^*, $[\![\phi]\!]_\rho^{\rho^*}$ is a valid logical proposition. Similarly, we define an interpretation function $[\![.]\!]$ for expressions, which takes the assignments ρ and ρ^* respectively for registers in $\mathcal{R}$ and $\mathcal{R}^*$. When an expression e does not contain registers in $\mathcal{R}^*$, e.g., when it is an expression of the programming language, we may simply write $[\![e]\!]_\rho$ instead of $[\![e]\!]_\rho^{\rho^*}$.

2.1.3 Verification Condition Generator

Verification condition generators (VCgens) produce the proof obligations that must be discharged in order to guarantee that the program meets its specifica-

tion. The predicate transformer wp is a partial function that computes, from a sufficiently annotated program, a fully annotated program in which all labels of the program have an explicit precondition attached to them. To ensure the computability of the wp function, its domain is restricted to *well-annotated* programs. This domain can be characterized by an inductive and decidable definition and does not impose any specific structure on programs.

Definition 2.1.

- *The graph of a function f is closed if for every node all its successors are in the graph:*
$$\mathsf{closed}(f) = \forall L \in G_f, \mathsf{succ}_f(L) \subseteq G_f$$
- *A label L' is reachable from a label L in f, if $L = L'$, or if it is the successor of a label reachable from L:*
$$L \in \mathsf{reachable}_{f,L}$$
$$L' \in \mathsf{reachable}_{f,L} \Rightarrow \forall L'' \in \mathsf{succ}_f(L'), L'' \in \mathsf{reachable}_{f,L}$$
- *A label L in a function f reaches annotated labels, if its associated instruction contains an assertion, or if its associated instruction is a* return *instruction (in that case the annotation is the post condition), or if all its immediate successors reach annotated labels. More precisely,* $\mathsf{reachAnnot}_f$ *is defined as the smallest set that satisfies the following conditions:*
$$f[L] = (\varphi,\ \mathsf{ins}) \Rightarrow L \in \mathsf{reachAnnot}_f$$
$$f[L] = \mathsf{return}\ r \Rightarrow L \in \mathsf{reachAnnot}_f$$
$$(\forall L' \in \mathsf{succ}_f(L), L' \in \mathsf{reachAnnot}_f) \Rightarrow L \in \mathsf{reachAnnot}_f$$
- *A function f is well annotated if it is closed and every reachable point from the starting point L_{sp} reaches annotated labels. A program p is well annotated if all its functions are well annotated.*

From the definition above, if every loop in the code graph of the function f is annotated, then f is well-annotated. It also follows from the definition above that, from a well-annotated program, one can compute an assertion for every label, i.e., the application of the function wp terminates for every program loop.

A fully-annotated function is computed from a partially annotated function f, using the wp_f transformer, using the preconditions and postconditions of functions called by f. The definition of $\mathsf{wp}_f(L)$ proceeds by case analysis: if L points to an instruction that carries an assertion ϕ, then $\mathsf{wp}_f(L)$ is set to ϕ; otherwise, $\mathsf{wp}_f(L)$ is computed by the function $\mathsf{wp}_f^{\mathtt{id}}$.

The formal definitions of wp_f and $\mathsf{wp}_f^{\mathtt{id}}$ are given in Figure 2.4, where the expression $\phi[{}^{e}/_{r}]$ stands for the substitution in the assertion ϕ of all occurrences of register r by the expression e. The definition of $\mathsf{wp}_f^{\mathtt{id}}$ is standard for assignment and conditional jumps, where $\langle\mathsf{op}\rangle$ and $\langle\mathsf{cmp}\rangle$ is the obvious interpretation of operators in RTL into expressions in the language of assertions.

$$
\begin{aligned}
\mathsf{wp}_f(L) &= \varphi && \text{if } G_f(L) = (\varphi,\ \mathsf{ins})\\
\mathsf{wp}_f(L) &= \mathsf{wp}_f^{\mathtt{id}}(\mathsf{ins}) && \text{if } G_f(L) = \mathsf{ins}\\
\mathsf{wp}_f^{\mathtt{id}}(r_d := \mathsf{op},\ L) &= \mathsf{wp}_f(L)[{}^{\langle \mathsf{op}\rangle}\!/_{r_d}]\\
\mathsf{wp}_f^{\mathtt{id}}(r_d := g(\vec{r}),\ L) &= \mathsf{pre}(g)[{}^{\vec{r}}\!/_{\vec{r}_g}]\\
&\qquad \wedge(\forall \mathsf{res}.\ \mathsf{post}(g)[{}^{\vec{r}}\!/_{\vec{r}^*_g}] \Rightarrow \mathsf{wp}_f(L)[{}^{\mathsf{res}}\!/_{r_d}])\\
\mathsf{wp}_f^{\mathtt{id}}(\mathsf{cmp}\ ?\ L_t : L_f) &= (\langle \mathsf{cmp}\rangle \Rightarrow \mathsf{wp}_f(L_t)) \wedge (\neg\langle \mathsf{cmp}\rangle \Rightarrow \mathsf{wp}_f(L_f))\\
\mathsf{wp}_f^{\mathtt{id}}(\mathsf{return}\ r) &= \mathsf{post}(f)[{}^{r}\!/_{\mathsf{res}}]\\
\mathsf{wp}_f^{\mathtt{id}}(\mathsf{nop},\ L) &= \mathsf{wp}_f(f[L])
\end{aligned}
$$

Fig. 2.4. Verification condition generator

For a function invocation, $\mathsf{wp}_f^{\mathtt{id}}(r_d := g(\vec{r}),\ L)$ is defined as a conjunction of the precondition of g, where formal parameters are replaced by actual parameters, and of the assertion $\forall \mathsf{res}.\ \mathsf{post}(g)[{}^{\vec{r}}\!/_{\vec{r}^*_g}] \Rightarrow \mathsf{wp}_f(L)[{}^{\mathsf{res}}\!/_{r_d}]$. The second conjunct permits that information in $\mathsf{wp}_f(L)$ about registers different from r_d be propagated to other preconditions. In the remainder of the chapter, we shall abuse notation and write $\mathsf{wp}_f^{\mathtt{id}}(L)$ instead of $\mathsf{wp}_f^{\mathtt{id}}(\mathsf{ins})$ if $f[L] = \mathsf{ins}$.

2.1.4 Certified Programs

Certificates provide a formal representation of proofs, and are used to verify that the proof obligations generated by the VCgen hold. For the purpose of certificate translation, we do not need to commit to a specific format for certificates. Instead, we assume that certificates are closed under specific operations on certificates, which are captured by an abstract notion of proof algebra.

Recall that a judgment is a pair consisting of a list of assertions, called context, and of an assertion, called goal. A proof algebra is given by a set-valued function $\mathcal{C}$ over judgments, and by a set of operations, all implicitly quantified in the usual way. The operations are standard (given in Figure 2.5), to the exception perhaps of the substitution operator that allows one to substitute a subexpression e by e' from the hypothesis $e = e'$, and of the operator ring, which establishes all ring equalities that will be used to justify the optimizations.

In order to remain at an abstract level, we do not provide an algorithm for checking certificates. Instead, we take $\mathcal{C}(\Gamma \vdash \varphi)$ to be the set of valid certificates of the judgment $\Gamma \vdash \varphi$. In the sequel, we write $\lambda : \Gamma \vdash \varphi$ to express that λ is a valid certificate for $\Gamma \vdash \varphi$, and use proof as a synonym of valid certificate. Furthermore, we require the certificate infrastructure to be sound, i.e., if $\mathcal{C}(\Gamma \vdash \phi) \neq \emptyset$ then for all maps ρ, ρ^*, if for every $\psi \in \Gamma$, $[\![\psi]\!]_\rho^{\rho^*}$ is valid, then $[\![\phi]\!]_\rho^{\rho^*}$ is valid.

$$
\begin{array}{ll}
\mathsf{intro}_{\mathsf{true}} & : \mathcal{C}(\Gamma \vdash \mathsf{true}) \\
\mathsf{axiom}\ A & : \mathcal{C}(\Gamma \vdash A) \qquad \text{if } A \in \Gamma \\
\mathsf{ring} & : \mathcal{C}(\Gamma \vdash n_1 = n_2) \quad \text{if } n_1 = n_2 \text{ is a ring equality} \\
\mathsf{intro}_{\wedge} & : \mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma \vdash B) \to \mathcal{C}(\Gamma \vdash A \wedge B) \\
\mathsf{elim}^{\mathsf{l}}_{\wedge} & : \mathcal{C}(\Gamma \vdash A \wedge B) \to \mathcal{C}(\Gamma \vdash A) \\
\mathsf{elim}^{\mathsf{r}}_{\wedge} & : \mathcal{C}(\Gamma \vdash A \wedge B) \to \mathcal{C}(\Gamma \vdash B) \\
\mathsf{intro}_{\Rightarrow} & : \mathcal{C}(\Gamma; A \vdash B) \to \mathcal{C}(\Gamma \vdash A \Rightarrow B) \\
\mathsf{elim}_{\Rightarrow} & : \mathcal{C}(\Gamma \vdash A \Rightarrow B) \to \mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma \vdash B) \\
\mathsf{elim}_{=} & : \mathcal{C}(\Gamma \vdash e_1 = e_2) \to \mathcal{C}(\Gamma \vdash A[{}^{e_1}\!/_{r}]) \to \mathcal{C}(\Gamma \vdash A[{}^{e_2}\!/_{r}]) \\
\mathsf{subst}\ r\ e & : \mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma[{}^{e}\!/_{r}] \vdash A[{}^{e}\!/_{r}]) \\
\mathsf{weak}\ \Delta & : \mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma; \Delta \vdash A) \\
\mathsf{intro}_{\forall} & : \mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma \vdash \forall r.A) \qquad \text{if } r \text{ is not in } \Gamma \\
\mathsf{elim}_{\forall} & : \mathcal{C}(\Gamma \vdash \forall r.A) \to \mathcal{C}(\Gamma \vdash A)
\end{array}
$$

Fig. 2.5. Proof Algebra

Finally, we define a certified program as one whose functions are certified, i.e., carry valid certificates for the proof obligations attached to them.

Definition 2.2.

- *A function f with declaration $\{\vec{r};\ \varphi;\ G;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ is certified if:*
  - *λ is a proof of $\vdash \varphi \Rightarrow \mathsf{wp}_f(L_{\mathsf{sp}})[{}^{\vec{r}}\!/_{\vec{r}^*}]$,*
  - *$\vec{\Lambda}(L)$ is a proof of $\vdash \varphi \Rightarrow \mathsf{wp}^{\mathtt{id}}_f(\mathsf{ins})$ for all reachable labels L in f such that $f[L] = (\varphi,\ \mathsf{ins})$.*
- *A program is certified if all its functions are.*

2.1.5 Soundness of PCC Infrastructure

The verification condition generator is sound, in the sense that if a certified program p is called with registers set to values that verify the precondition of the function main, and it terminates normally, then the final state will verify the postcondition of main.

When considering mutually-recursive functions, special care must be taken to ensure soundness of the VCgen. In this case it is not hard to achieve since we are only interested in verifying the partial correctness of the program w.r.t. its specification, i.e., we only consider finite executions.

Lemma 2.3. *Let p be a certified program. Then, for every function f with declaration $\{\vec{r};\ \varphi;\ G;\ \psi;\ \lambda;\ \vec{\Lambda}\}$, any initial mapping ρ^* with domain $\{r_1^*, \ldots, r_k^*\}$, any label L in the domain of G_f and any state ρ, if $[\![\mathsf{wp}_f(L)]\!]^{\rho^*}_{\rho}$ and $\langle f[L], \rho\rangle \rightsquigarrow_f n$ then $[\![\psi]\!]^{\rho^*}_{[\mathsf{res}\mapsto n]}$.*

Proof. Since wp *and* $\mathsf{wp}^{\mathsf{ins}}$ *are defined each one in terms of the other, we prove the goal of the lemma above simultaneously with a similar goal but under the hypothesis* $[\![\mathsf{wp}^{\mathsf{ins}}_f(L)]\!]^{\rho^*}_{\rho}$. *The proof proceeds by rule induction on the derivation of* $\langle f[L], \rho\rangle \rightsquigarrow_{\mathcal{F}} n$. *For simplicity, we rely on the following standard results (where* $FV(\varphi)$ *stands for the set of unbound variables in* φ*):*

i) **(Coincidence Lemma)**. *For all states* $\rho_1, \rho_2, \rho_1^*, \rho_2^*$ *and assertion* φ, *if for all* x *in* $FV(\varphi) \cap \mathcal{R}$ *and* y *in* $FV(\varphi) \cap \mathcal{R}^*$ *we have* $\rho_1 x = \rho_2 x$ *and* $\rho_1^* y = \rho_2^* y$ *then* $[\![\varphi]\!]^{\rho_1^*}_{\rho_1} = [\![\varphi]\!]^{\rho_2^*}_{\rho_2}$.

ii) **(Substitution Lemma)**. *For all* x, c, φ *and* ρ, ρ^*, $[\![\phi[{}^{e}/_{x}]]\!]^{\rho^*}_{\rho} = [\![\phi]\!]^{\rho^*}_{[\rho|x\mapsto [\![e]\!]^{\rho^*}_{\rho}]}$ *and* $[\![\phi[{}^{e}/_{x^*}]]\!]^{\rho^*}_{\rho} = [\![\phi]\!]^{[\rho^*|x^*\mapsto [\![e]\!]^{\rho^*}_{\rho}]}_{\rho}$.

- *Consider the case s.t. the last rule applied is*

$$\frac{\langle \mathsf{ins}, \rho\rangle \rightsquigarrow_{\mathcal{F}} n}{\langle(\varphi,\ \mathsf{ins}), \rho\rangle \rightsquigarrow_{\mathcal{F}} n}$$

  then $\mathsf{wp}_f(L) = \phi$ *and since* f *is certified, and the certificate infrastructure is sound, we have that* $[\![\phi \Rightarrow \mathsf{wp}^{\mathsf{ins}}_f(L)]\!]^{\rho^*}_{\rho}$ *is valid. Hence, from the hypothesis* $[\![\mathsf{wp}_f(L)]\!]^{\rho^*}_{\rho}$ *and definition of* $[\![.]\!]$, *we have that* $[\![\mathsf{wp}^{\mathsf{ins}}_f(L)]\!]^{\rho^*}_{\rho}$ *is valid. By I.H. and the latter condition we get* $[\![\psi_f]\!]^{\rho^*}_{[\mathsf{res}\mapsto n]}$. *Notice that this is the only case where we need to distinguish the hypothesis* $[\![\mathsf{wp}^{\mathsf{ins}}(L)]\!]^{\rho^*}_{\rho}$ *from* $[\![\mathsf{wp}(L)]\!]^{\rho^*}_{\rho}$*; in any other case* $\mathsf{wp}(L) = \mathsf{wp}^{\mathsf{ins}}(L)$.
- *Assume the last rule applied is*

$$\frac{\langle f[L'], [\rho \mid r_d \mapsto [\![\mathsf{op}]\!]^{\rho^*}_{\rho}]\rangle \rightsquigarrow_{\mathcal{F}} n}{\langle r_d := \mathsf{op},\ L', \rho\rangle \rightsquigarrow_{\mathcal{F}} n}\ .$$

  By hypothesis and definition of wp_f, *we have* $[\![\mathsf{wp}_f(L')[{}^{\mathsf{op}}/_{r_d}]]\!]^{\rho^*}_{\rho}$. *Equivalently, by substitution lemma* $[\![\mathsf{wp}_f(L')]\!]^{\rho^*}_{[\rho|r_d\mapsto [\![\mathsf{op}]\!]^{\rho^*}_{\rho}]}$
- *Assume the last rule applied involves a function call, i.e., there is a function* g *s.t.* $f[L] = ret := g(\vec{r}),\ L'$ *and the last rule applied is*

$$\frac{\langle g[L_{\mathsf{sp}}], [\vec{r}_g \mapsto \rho\vec{r}]\rangle \rightsquigarrow_{g} m \quad \langle f[L'], [\rho \mid ret \mapsto m]\rangle \rightsquigarrow_{\mathcal{F}} n}{\langle ret := g(\vec{r}),\ L', \rho\rangle \rightsquigarrow_{\mathcal{F}} n}$$

  for some value m. *Let* φ_g *and* ψ_g *stand for the preconditions and postcondition of* g *respectively. From* $[\![\mathsf{wp}_f(L)]\!]^{\rho^*}_{\rho}$ *and definition of* $[\![.]\!]$, *we have* $[\![\varphi_g[{}^{\vec{r}}/_{\vec{r}_g}]]\!]^{\rho^*}_{\rho}$. *By coincidence lemma we have then* $[\![\varphi_g[{}^{\vec{r}}/_{\vec{r}_g}]]\!]^{[r_g^*\mapsto \rho r]}_{[r\mapsto \rho r]}$, *and by substitution lemma* $[\![\varphi_g]\!]^{[r_g^*\mapsto \rho r]}_{[r_g\mapsto \rho r]}$. *Since* g *is certified, we have a proof for* $\varphi_g \Rightarrow \mathsf{wp}_g(L_{sp})[{}^{\vec{r}_g}/_{\vec{r}_g^*}]$ *and therefore* $[\![\mathsf{wp}_g(L_{sp})[{}^{\vec{r}_g}/_{\vec{r}_g^*}]]\!]^{[r_g^*\mapsto \rho r]}_{[r_g\mapsto \rho r]}$.

Again by substitution lemma, $[\![\mathsf{wp}_g(L_{sp})]\!]^{[r_g^* \mapsto \rho r]}_{[r_g \mapsto \rho r]}$. *Therefore, by application of I.H., we know that* $[\![\psi_g]\!]^{[r_g^* \mapsto \rho r]}_{[\mathsf{res} \mapsto m]}$, *and then by substitution lemma* $[\![\psi_g[{}^{m}/_{\mathsf{res}}]]\!]^{[r_g^* \mapsto \rho r]}_{\rho}$. *From the hypothesis* $[\![\mathsf{wp}_f(L)]\!]^{\rho^*}_{\rho}$ *and definition of* $[\![.]\!]$, *we have that* $[\![\psi_g[{}^{\vec{r}}/_{\vec{r_g^*}}]]\!]^{\rho^*}_{[\rho|\mathsf{res} \mapsto m]} \Rightarrow [\![\mathsf{wp}_f(L')[{}^{\mathsf{res}}/_{ret}]]\!]^{\rho^*}_{[\rho|\mathsf{res} \mapsto m]}$. *Equivalently by substitution lemma,* $[\![\psi_g[{}^{m}/_{\mathsf{res}}]]\!]^{[r_g^* \mapsto \rho r]}_{\rho} \Rightarrow [\![\mathsf{wp}_f(L')]\!]^{\rho^*}_{[\rho|ret \mapsto m]}$, *and hence* $[\![\mathsf{wp}_f(L')]\!]^{\rho^*}_{[\rho|ret \mapsto m]}$. *From the latter condition and I.H., we get the desired result.*

As a corollary, we obtain the following theorem:

Theorem 2.4 (Soundness of VCgen). *Suppose that*

a) P *is a certified program containing a function* main, *with precondition* Φ *and postcondition* Ψ,
b) ρ *is such that* $[\![\Phi]\!]_\rho$, *and*
c) $\langle \mathsf{main}[L_{sp}], \rho \rangle \rightsquigarrow n$,

then the interpretation $[\![\Psi]\!]^{[\vec{r^*} \mapsto \rho \vec{r}]}_{[\mathsf{res} \mapsto n]}$ *is valid.*

2.2 Preservation of Proof Obligations

The purpose of this section is to establish preservation of proof obligations for a nonoptimizing compiler from an imperative language with procedures to RTL. This result is inspired from earlier work by Barthe, Rezk, and Saabas [14].

In this section, we define a simple and structured high-level language, a standard verification condition generator for this language, and a nonoptimizing compiler to the RTL language defined before. Then, we show that given the same program specification, proof obligations for the source program and for its compiled RTL version coincide.

2.2.1 Source Language

A program p in the source language is defined as a function from function identifiers to function declarations. We assume that every program comes equipped with a special function identifier, namely main, and its declaration. The declaration of a function f in the source language has the form: $\{\vec{x_f};\ \varphi;\ c;\ \psi;\ \lambda;\ \vec{\Lambda}\}$, where c is a command whose syntax is shown in Figure 2.6. Every function returns integer values.

As in RTL programs, a function declaration includes its formal parameters $\vec{x}$, a precondition φ, a postcondition ψ, a certificate λ, and a *set* of certificates $\vec{\Lambda}$. The source language features the same annotation language as RTL.

$$
\begin{array}{l}
\star ::= < \mid \leq \mid = \mid \geq \mid > \mid \wedge \mid \vee \\
e ::= x \mid n \mid -e \mid e \ + \ e \mid e \ - \ e \mid e \ * \ e \\
c ::= \texttt{skip} \mid x := e \mid c; c \mid \texttt{while } \{\phi\} \ e \star e \ \texttt{do } c \mid \\
\qquad \texttt{if } e \star e \ \texttt{then } c \ \texttt{else } c \mid y := \texttt{call } f(\vec{x}) \mid \\
\qquad \texttt{return } e
\end{array}
$$

Fig. 2.6. Syntax of source language

However, the only command that has an annotation is the `while` command. Notice that all `while` commands hold annotations. The definition of the VC-gen is made in terms of the function wp, which is overloaded to denote as well a predicate transformer for the source code, and is given in Figure 2.7. Certificates for source level and RTL programs are represented by the same proof algebra.

$$
\begin{array}{ll}
\mathsf{wp}(\texttt{skip}, \psi) & = \psi \\
\mathsf{wp}(x := e, \psi) & = \psi[e/x] \\
\mathsf{wp}(c_1; c_2, \psi) & = \mathsf{wp}(c_1, \mathsf{wp}(c_2, \psi)) \\
\mathsf{wp}(\texttt{while } \{\varphi\} \ e_1 \star e_2 \ \texttt{do } c_1, \psi) & = \varphi \\
\mathsf{wp}(\texttt{if } e_1 \star e_2 \ \texttt{then } c_1 \ \texttt{else } c_2, \psi) & = (\langle e_1 \star e_2 \rangle \Rightarrow \mathsf{wp}(c_1, \psi)) \wedge \\
 & \quad (\neg \langle e_1 \star e_2 \rangle \Rightarrow \mathsf{wp}(c_2, \psi)) \\
\mathsf{wp}(y := \texttt{call } g(\vec{x}), \psi) & = \mathsf{pre}(g)[{}^{\vec{x}}/_{\vec{x}_g}] \wedge \\
 & \quad \forall \mathsf{res}.(\mathsf{post}(g)[{}^{\vec{x}}/_{\vec{x}^*_g}] \Rightarrow \psi[{}^{\mathsf{res}}/_{y}]) \\
\mathsf{wp}(\texttt{return } e, \psi) & = \psi[{}^{e}/_{\mathsf{res}}]
\end{array}
$$

Fig. 2.7. wp for the source language

$$
\begin{array}{ll}
\mathsf{PO}(\texttt{skip}, \psi) & = \emptyset \\
\mathsf{PO}(x := e, \psi) & = \emptyset \\
\mathsf{PO}(c_1; c_2, \psi) & = \mathsf{PO}(c_2, \psi) \cup \mathsf{PO}(c_1, \mathsf{wp}(c_2, \psi)) \\
\mathsf{PO}(\texttt{while } \{\phi\} \ e_1 \star e_2 \ \texttt{do } c_1, \psi) & = \\
\qquad \mathsf{PO}(c_1, \phi) \cup \{\phi \Rightarrow (e_1 \star e_2 \Rightarrow \mathsf{wp}(c_1, \phi)) \wedge (\neg(e_1 \star e_2) \Rightarrow \psi)\} & \\
\mathsf{PO}(\texttt{if } e_1 \star e_2 \ \texttt{then } c_1 \ \texttt{else } c_2, \psi) & = \mathsf{PO}(c_1, \psi) \cup \mathsf{PO}(c_2, \psi) \\
\mathsf{PO}(y := \texttt{call } g(\vec{x}), \psi) & = \emptyset \\
\mathsf{PO}(\texttt{return } e, \psi) & = \emptyset
\end{array}
$$

Fig. 2.8. Proof obligations for the source language

Definition 2.5.

- *A function f with declaration $\{\vec{x};\ \varphi;\ c;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ is certified if:*
  - *λ is a proof of $\vdash \varphi \Rightarrow \mathsf{wp}(c,\psi)[\vec{r}/\vec{x^*}]$*
  - *$\vec{\Lambda}$ contains a proof of $\vdash \varphi$ for every proof obligation φ in $\mathsf{PO}(c,\psi)$, where PO is defined in Figure 2.8.*
- *A program is certified if all its functions are.*

The semantics of the source language is standard and, thus, omitted.

2.2.2 Compilation

The compilation function to RTL is standard, except for the `while` command, and sketched in Figure 2.9. Note that the compilation function $[\![.]\!]$ takes, in addition to the command to compile, two additional parameters. Both are labels and respectively correspond to the label of the first instruction in the compilation and the label of the successor of the last instruction. For clarity, Figure 2.9 is just a scheme of the compiler because the compiler for expressions $[\![.]\!]$ is using a unique instruction $L_B :\ r_1 := [\![e_1]\!],\ L$, while compilation of the expression e_1 might need more than one RTL instruction, and r_1 will be assigned with the result of the last register assigned to in the compiled instructions for e_1.

$$
\begin{aligned}
[\![\texttt{while}\ \{\varphi\}\ e_1 \star e_2\ \texttt{do}\ c]\!]_{L_H,L_E} = {} & L_H\ :\ (\varphi,\ \mathsf{nop},\ L_B)\\
& L_B\ :\ r_2 := [\![e_2]\!],\ L'_B\\
& L'_B\ :\ r_1 := [\![e_1]\!],\ L\\
& L\ :\ r_1 \star r_2\ ?\ L_T : L_E\\
& [\![c]\!]_{L_T,L_H}\\
[\![\texttt{skip}]\!]_{L,L'} = {} & L\ :\ \mathsf{nop},\ L'\\
[\![x := e]\!]_{L,L'} = {} & L\ :\ x := [\![e]\!],\ L'\\
[\![c_1; c_2]\!]_{L,L'} = {} & [\![c_1]\!]_{L,L''}\\
& [\![c_2]\!]_{L'',L'}\\
[\![\texttt{if}\ e \star e\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!]_{L,L'} = {} & L\ :\ r_1 := [\![e_1]\!],\ L_1\\
& L_1\ :\ r_2 := [\![e_2]\!],\ L_2\\
& L_2\ :\ r_1 \star r_2\ ?\ L_T : L_F\\
& [\![c_1]\!]_{L_T,L'}\\
& [\![c_2]\!]_{L_F,L'}\\
[\![y := \texttt{call}\ f(\vec{x})]\!]_{L,L'} = {} & L\ :\ y := f(\vec{x}),\ L'
\end{aligned}
$$

Fig. 2.9. Compiler definition

2.2.3 Preservation of Proof Obligations

The wp of a source code function is syntactically equivalent to the wp of its compilation, provided the variables of the source language and the registers of RTL are equivalent. For notational convenience, in the following proofs we let the expression $f[L, L']$ stand for the subgraph of nodes reachable from label L without traversing (and not including) the node at label L'.

Lemma 2.6. *Let $f[L, L']$ be a subgraph code of the RTL function f given by compilation $[\![c]\!]_{L,L'}$ of a command c. Then, $\mathsf{wp}(c, \psi) = \mathsf{wp}_f(L)$, where $\psi = \mathsf{wp}_f(L')$.*

Proof. The proof proceeds by structural induction on the command c. For simplicity, we assume the following result about the compilation of expressions:

$$\text{if } f[l, l'] = r := [\![e]\!]_{L,L'} \text{ then } \mathsf{wp}_f(l) = \mathsf{wp}_f(l')[^e/_r] \tag{2.1}$$

- *case $c =$ `while` $\{\varphi\}$ $e_1 \star e_2$ `do` c. In this case $\mathsf{wp}(c, \psi)$ is equal to ϕ, as well as $\mathsf{wp}_f(L)$ by definition of $[\![.]\!]$.*
- *case $c = x := e$. We have that $\mathsf{wp}(c, \psi)$ is equal to $\psi[^e/_x]$, and thus, to $\mathsf{wp}_f(L')[^e/_x]$ by hypothesis. From definition of $[\![.]\!]$, $f[L, L'] = r := [\![e]\!]_{L,L'}$, and property (2.1), we have that $\mathsf{wp}(c, \psi)$ is equal to $\mathsf{wp}_f(L)$.*
- *case $c = c_1; c_2$. By definition of wp, $\mathsf{wp}(c, \psi) = \mathsf{wp}(c_1, \mathsf{wp}(c_2, \psi))$. By definition of $[\![c]\!]_{L,L'}$ we have $f[L, L''] = [\![c_1]\!]_{L,L''}$ and $f[L'', L'] = [\![c_2]\!]_{L'',L'}$, then, by I.H. $\mathsf{wp}(c_2, \psi) = \mathsf{wp}_f(L'')$. Hence, $\mathsf{wp}(c, \psi) = \mathsf{wp}(c_1, \psi')$ where $\psi' = \mathsf{wp}_f(L'')$ and, since again by I.H. we have $\mathsf{wp}(c_1, \psi') = \mathsf{wp}_f(L)$, we get $\mathsf{wp}(c, \psi) = \mathsf{wp}_f(L)$.*

Hence, one can prove that proof obligations and certificates are preserved[2] along nonoptimizing compilation.

Lemma 2.7. *Let f with declaration $\{\vec{x};\ \varphi;\ c;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ be a certified source function. Then $\overline{f}$ declared as $\{\vec{x};\ \varphi;\ [\![c]\!];\ \psi;\ \lambda;\ \vec{\Lambda}\}$ is a certified RTL function.*

Proof. The proof consists on verifying that f and $\overline{f}$ contain exactly the same proof obligations. To this end, consider a subprogram c s.t. $\overline{f}[l, l'] = [\![c]\!]_{l,l'}$ and $\psi = \mathsf{wp}_{\overline{f}}(l')$. Then, we show by structural induction on c, that the proof obligations induced by assertions in $\overline{f}[l, l']$ correspond to the proof obligations in $\mathsf{PO}(c, \psi)$. We only consider the case $c =$ `while` $\{\phi\}$ $e_1 \star e_2$ `do` c'. The proof obligations in c w.r.t. ψ are the proof obligations in $\mathsf{PO}(c', \phi)$ plus

$$\phi \Rightarrow (e_1 \star e_2 \Rightarrow \mathsf{wp}(c', \phi)) \wedge (\neg(e_1 \star e_2) \Rightarrow \psi) \ .$$

By definition of $[\![.]\!]$ we have

[2] Strictly speaking, the first $\vec{\Lambda}$ in the source program is a set, whereas the second $\vec{\Lambda}$ in the compiled program is a map, but it is immediate to turn one into the other.

$$
\begin{array}{rl}
f[l,l'] = & l \ : \ (\phi,\ \mathsf{nop},\ l_b) \\
& l_b \ : \ r_2 := [\![e_2]\!],\ l'_b \\
& l'_b \ : \ r_1 := [\![e_1]\!],\ l'' \\
& l'' \ : \ r_1 \star r_2 \ ?\ l_T : l' \\
& [\![c]\!]_{l_T,l}
\end{array}
$$

Annotations in $f[l,l']$ are ϕ plus the annotations in c'. By I.H., proof obligations in $f[l_T,L]$ are exactly the proof obligations in $\mathsf{PO}(c',\phi)$. The annotation ϕ in l induces the proof obligation $\phi \Rightarrow \mathsf{wp}^{\mathsf{ins}}_f(l_B)$, that after unfolding of wp_f and $\mathsf{wp}^{\mathsf{ins}}_f$ can be shown equal to

$$\phi \Rightarrow (e_1 \star e_2 \Rightarrow \mathsf{wp}_f(l_T)) \wedge (\neg(e_1 \star e_2) \Rightarrow \mathsf{wp}_f(l'))$$

which, by Lemma 2.6, is equal to

$$\phi \Rightarrow (e_1 \star e_2 \Rightarrow \mathsf{wp}(c',\phi)) \wedge (\neg(e_1 \star e_2) \Rightarrow \psi) \ .$$

We have seen in this section that the first phase of a compiler, that translates a high-level structured program into an RTL representation, preserves verification conditions if no optimization is applied. In the next section we extend this simple compiler with standard optimization phases and for each of them we propose a transformation of the certificate.

2.3 Certificate Translation for Common Optimizations

This section provides instances of certificate translation for common RTL optimizations. The order of optimizations is chosen for the clarity of exposition and does not necessarily reflect the order in which the optimizations are performed by a compiler.

2.3.1 Overview

In a classical compiler, transformations operate on unannotated programs, and are performed in two phases: first, a data flow analysis gathers information about the program. Then, on the basis of this information, (blocks of) instructions are rewritten. Consider for example the following annotated piece of code:

$$
\begin{array}{rl}
& \{\mathsf{true}\} \\
& r_1 := n \\
L: & \{r_1 \geq n\},\ L' \\
L': & r_2 := r_1 \\
& \{r_1 = r_2\}
\end{array}
$$

An analysis may detect, ignoring annotations, that the register r_1 always stores the value n at program points L and L'. Later, a transformation phase

optimizes the code replacing the assignment $r_2 := r_1$ by the more efficient $r_2 := n$.

$$\begin{array}{rl} & \{\mathsf{true}\} \\ & r_1 := n \\ L: & \{r_1 \geq n\},\ L' \\ L': & r_2 := n \\ & \{r_1 = r_2\} \end{array}$$

According to Definition 2.2, a certificate for an optimized function $\bar{f}$ must include a proof that the precondition of $\bar{f}$ implies the precondition of its first instruction, and a proof, for each label L of $\bar{f}$, that the assertion at L implies the precondition of instruction L. In the example, the proof obligations corresponding to the original fragment of code are $\mathsf{true} \Rightarrow n \geq n$ and $r_1 \geq n \Rightarrow r_1 = r_1$. After the transformation we have that proof obligations are $\mathsf{true} \Rightarrow n \geq n$ and $r_1 \geq n \Rightarrow n = r_1$. Not only does one of the proof obligations not coincide with the original one (and hence the original certificate cannot be reused), but it also becomes unprovable.

Remark. The difficulty shown in the example above does not apply to more complex VCgens that propagates proof obligations backwards. In the example, such VCgen computes the wp of the statement $r_1 := n$ over the proof obligations $r_1 \geq n \Rightarrow r_1 = r_1$ and $r_1 \geq n \Rightarrow n = r_1$, returning $n \geq n \Rightarrow n = n$ and $n \geq n \Rightarrow n = n$, respectively. However, the following example

$$\begin{array}{rl} & r_1 := 1 \\ L_o: & \{\mathsf{true}\} \\ & r_1 < n\ ?\ L : L' \\ L: & n := n + 1,\ L_o \\ L': & r_2 := r_1 \\ & \{r_1 = r_2\} \end{array}$$

shows that a more complex VCgen does not remove the problem explained above since, indeed, the program becomes unprovable when the assignment $r_2 := r_1$ is substituted by $r_2 := n$. Hence, since we do not have a fundamental reason to prefer one VCgen to the other one, we opt for a basic VCgen to simplify the description of certificate translators. *End of Remark.*

The above example illustrates that the validity of annotations is not necessarily preserved by program transformations. In order to maintain their validity, many optimizations require strengthening the original annotations by assertions that capture in a logical form the results of the analysis that underlies the optimization. Intuitively, the need to strengthen assertions stems from the fact that semantic preservation of program transformations must eventually be justified by the conditions returned by the analysis.

Therefore, optimized programs are defined by augmenting annotations with the information returned by the analysis, expressed as an assertion and denoted $\mathrm{RES}_{\mathcal{A}}(L)$ below.

Definition 2.8. *The optimized graph code of a function f is defined as follows:*

$$G_{\bar{f}}(L) = \begin{cases} (\varphi \wedge \mathrm{RES}_{\mathcal{A}}(L),\ [\![\mathsf{ins}]\!]) & \text{if } G_f(L) = (\varphi,\ \mathsf{ins}) \\ [\![\mathsf{ins}]\!] & \text{if } G_f(L) = \mathsf{ins} \end{cases}$$

where $[\![\mathsf{ins}]\!]$ is the optimized version of instruction ins*. In the sequel, we write $\bar{\varphi}_L$ for $\varphi_L \wedge \mathrm{RES}_{\mathcal{A}}(L)$.*

(Note that the above definition is restricted to optimizations that do not modify the graph topology, such as constant propagation, common subexpression elimination or induction variable strength reduction).

Augmenting an assertion ϕ into $\phi \wedge \mathrm{RES}_{\mathcal{A}}(L)$ has two immediate effects. On the negative side, the inserted condition $\mathrm{RES}_{\mathcal{A}}(L)$ requires certificates for the new proof obligations involving the results of the analysis. To solve this issue, an automatic procedure for certification of the analysis, namely a certifying analyzer, is applied as a first step, producing the certified program

$$f_{\mathcal{A}} = \{\vec{r}_f;\ \mathsf{true};\ G_{\mathcal{A}};\ \mathsf{true};\ \lambda_{\mathcal{A}};\ \vec{\Lambda}_{\mathcal{A}}\}$$

where $G_{\mathcal{A}}$ is a new version of G_f annotated with the results of the analysis, i.e., G_f such that $G_{\mathcal{A}}(L) = (\mathrm{RES}_{\mathcal{A}}(L),\ \mathsf{ins})$ for all labels L in f.

On the positive side, strengthening the antecedent enables us to build a proof for the transformed proof obligations. We illustrate the benefits of strengthening assertions on the example above, and then elaborate on the transformation of certificates.

Considering the example again, let us add the assertion $r_1 = n$, used to justify the transformation, at program point L. Then, the transformed program is suitably annotated as:

$$\begin{array}{ll} & \{\mathsf{true}\} \\ & r_1 := n \\ L: & \{r_1 \geq n \wedge r_1 = n\},\ L' \\ L': & r_2 := n \\ & \{r_1 = r_2\} \end{array}$$

In this case the proof obligation $r_1 \geq n \wedge r_1 = n \Rightarrow n = r_1$ is provable, but still does not coincide with the original. In such a simple case, one could generate a certificate for the proof obligation without using the certificate of the original proof obligation, but in the general case we will need to build a new certificate from the certificate of the original proof obligation (here $r_1 \geq n \Rightarrow r_1 = r_1$).

A systematic approach to generate certificates for $\bar{f}$ is to define two functions that transform the certificates for f:

$$\begin{array}{ll} T_0: & \mathcal{C}(\vdash \mathsf{pre}(f) \Rightarrow \mathsf{wp}_f(L_{\mathsf{sp}})[\vec{r}/_{\vec{r}^*}]) \rightarrow \mathcal{C}(\vdash \mathsf{pre}(\bar{f}) \Rightarrow \mathsf{wp}_{\bar{f}}(L_{\mathsf{sp}})[\vec{r}/_{\vec{r}^*}]) \\ T_\lambda: & \forall L,\ \mathcal{C}(\vdash \varphi_L \Rightarrow \mathsf{wp}^{\mathtt{id}}_f(L)) \rightarrow \mathcal{C}(\vdash \bar{\varphi}_L \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\bar{f}}(L)) \end{array}$$

where φ_L is the original assertion at label L, and $\bar{\varphi}_L$ is the augmented assertion at label L. Here the function T_0 transforms the proof that the precondition

implies the assertion at program point L_{sp} for f into a proof of the same fact for $\bar{f}$, and likewise, the function T_λ transforms for each reachable annotated label L the proof that its annotation implies the precondition at program point L for f into a proof of the same fact for $\bar{f}$.

The functions T_0 and T_λ can be constructed, independently of the optimization considered, from a function

$$T_L^{\mathsf{ins}} : \mathcal{C}(\vdash \mathsf{wp}_f^{\mathtt{id}}(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}_{\overline{f}}^{\mathtt{id}}(L))$$

that associates to every program point L in f, a proof of the following fact: the original annotation of f (i.e., $\mathsf{wp}_f^{\mathtt{id}}(L)$) and the hypothesis obtained from the results of the analysis (i.e., $\mathrm{RES}_{\mathcal{A}}(L)$) imply the annotation of $\overline{f}$ (i.e., $\mathsf{wp}_{\overline{f}}^{\mathtt{id}}(L)$).

The function T_λ is defined using the function T_L^{ins} and the certificate of the analysis as shown in Figure 2.10. To define T_0, i.e., to generate $\overline{\lambda}$ from

Let $\Gamma = [\mathsf{wp}_{\overline{f}}(L)]$ in:
p_1:=$\mathsf{axiom}(\mathsf{wp}_{\overline{f}}(L)) : \Gamma \vdash \mathsf{wp}_{\overline{f}}(L)$
p_2:=$\mathsf{elim}^{\mathsf{l}}_{\wedge}(p_1) : \Gamma \vdash \mathsf{wp}_f(L)$
p_3:=$\mathsf{elim}^{\mathsf{r}}_{\wedge}(p_1) : \Gamma \vdash \mathsf{wp}_{f_A}(L)$
p_4:=$\mathsf{weak}\ \Gamma(\vec{\Lambda}(L)) : \Gamma \vdash \mathsf{wp}_f(L) \Rightarrow \mathsf{wp}_f^{\mathtt{id}}(L)$
p_5:=$\mathsf{elim}_{\Rightarrow}(p_2, p_4) : \Gamma \vdash \mathsf{wp}_f^{\mathtt{id}}(L)$
p_6:=$\mathsf{weak}\ \Gamma(T_L^{\mathsf{ins}}(L)) : \Gamma \vdash \mathsf{wp}_f^{\mathtt{id}}(L) \Rightarrow \mathsf{wp}_{f_A}(L) \Rightarrow \mathsf{wp}_{\overline{f}}^{\mathtt{id}}(L)$
p_7:=$\mathsf{elim}_{\Rightarrow}(p_5, p_6) : \Gamma \vdash \mathsf{wp}_{f_A}(L) \Rightarrow \mathsf{wp}_{\overline{f}}^{\mathtt{id}}(L)$
p_8:=$\mathsf{elim}_{\Rightarrow}(p_3, p_7) : \Gamma \vdash \mathsf{wp}_{\overline{f}}^{\mathtt{id}}(L)$
p_9:=$\mathsf{intro}_{\Rightarrow}(p_8) : \vdash \mathsf{wp}_{\overline{f}}(L) \Rightarrow \mathsf{wp}_{\overline{f}}^{\mathtt{id}}(L)$

Fig. 2.10. Definition of $T_\lambda(\vec{\Lambda}(L))$ from T_L^{ins}

λ (recall that $\overline{\lambda}$ is a proof of $\mathsf{pre}(\bar{f}) \Rightarrow \mathsf{wp}_{\overline{f}}(L_{\mathsf{sp}})[\vec{r_g}/\vec{r_g^*}]$), we reuse λ (i.e., a proof of $\mathsf{pre}(f) \Rightarrow \mathsf{wp}_f(L_{\mathsf{sp}})[\vec{r_g}/\vec{r_g^*}]$), since $\mathsf{pre}(f)$ implies $\mathsf{wp}_f(L_{\mathsf{sp}})[\vec{r_g}/\vec{r_g^*}]$, and instantiating T_L^{ins} to L_{sp} we get a predicate equivalent to $\mathsf{wp}_f(L_{\mathsf{sp}}) \Rightarrow \mathsf{wp}_{\overline{f}}(L_{\mathsf{sp}})$ (assuming that the analysis is computed from the trivial precondition true).

Whereas the definition of T_0 and T_λ are generic, the function T_L^{ins} must be defined for each program optimization. It turns out that for many program optimizations it is possible to inductively define T_L^{ins} using the definition of $T_{L_1}^{\mathsf{ins}}, \ldots, T_{L_k}^{\mathsf{ins}}$, where $\{L_1, \ldots, L_k\}$ are the successor program points for L. Due to the presence of loops, T_L^{ins} may be not well defined in the general case. However, we only consider well-annotated programs, and the definition of well-annotated programs induces an induction principle with annotated program labels as the base case.

In summary, the definition of a certificate translator for a program optimization requires one to define a certifying analyzer, and a function T_L^{ins} with suitable characteristics. In the rest of this section, we show how to define these for many common program optimizations.

2.3.2 Constant Propagation

Description

Constant propagation aims at minimizing run-time evaluation of expressions and access to registers with constant values. It relies on a data flow analysis that returns a function $\mathcal{A}$ with type $\mathcal{PP} \times \mathcal{R} \to \mathbb{Z}_\bot$ ($\mathcal{PP}$ denoting the set of program points) such that $\mathcal{A}(L, r) = n$ indicates that r holds the value n every time execution reaches the label L. If the analysis cannot infer that a register r holds a constant value at label L then we write $\mathcal{A}(L, r) = \bot$.

The definition of constant propagation over a function f can be found in Figure 2.11. Exploiting the information provided by $\mathcal{A}$, the optimization consists of replacing instructions that read registers by equivalent instructions that read constants. Furthermore, in the case of a conditional instruction, if the truth value of an integer comparison can be statically determined, it is replaced by a jump instruction to the corresponding branching point.

For example, if both arguments of an addition operation are known to be equal to n_1 and n_2, the operation is directly replaced by an immediate load of the integer n s.t. $n = n_1 + n_2$. If only one register is known to be equal to 0 the compiler replaces the addition operation by a move instruction. If one register is known but not equal to 0 then the compiler uses an immediate addition operation. Similar kind of optimizations are done for other arithmetic operations, but are not shown in Figure 2.11.

The optimized function $\overline{f}$ will be such that $\overline{f}[L] = \|f[L]\|_L$ for every label L in the domain of G_f. The transformation of an annotated instruction is defined as a transformation of the annotation and the transformation of the instruction descriptor.

Certifying analyzer

In this paragraph, we describe a certifying analyzer for constant propagation as an extension of the standard analysis algorithm. First, we attach to each reachable label L the assertion $\mathrm{RES}_{\mathcal{A}}(L)$:

$$\mathrm{RES}_{\mathcal{A}}(L) \equiv \bigwedge_{\mathcal{A}(L,r)\neq\bot} r = \mathcal{A}(L, r)$$

To derive a certificate for the analysis we must, for each reachable label L, generate a proof for the judgment

$$\vdash \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{f_{\mathcal{A}}}(L)$$

$$
\begin{aligned}
[\![(\varphi,\ \mathsf{ins})]\!]_L &= (\varphi \wedge \mathrm{RES}_{\mathcal{A}}(L),\ [\![\mathsf{ins}]\!]^{\mathtt{id}}_L)\\
[\![\mathsf{ins}]\!]_L &= [\![\mathsf{ins}]\!]^{\mathtt{id}}_L
\end{aligned}
$$

$$
\begin{aligned}
[\![r_d := \mathsf{op},\ L']\!]^{\mathtt{id}}_L &= r_d := [\![\mathsf{op}]\!]^{\mathtt{op}}_L,\ L'\\
[\![\mathsf{cmp}\ ?\ L_t : L_f]\!]^{\mathtt{id}}_L &= \begin{cases} \mathsf{nop},\ L_t \text{ when } [\![\mathsf{cmp}]\!]^{\mathtt{cmp}}_L = \mathsf{true}\\ \mathsf{nop},\ L_f \text{ when } [\![\mathsf{cmp}]\!]^{\mathtt{cmp}}_L = \mathsf{false}\\ [\![\mathsf{cmp}]\!]^{\mathtt{cmp}}_L\ ?\ L_t : L_f \quad \text{otherwise}\end{cases}\\
[\![\mathsf{ins}]\!]^{\mathtt{id}}_L &= \mathsf{ins} \qquad \text{in any other cases}
\end{aligned}
$$

$$
\begin{aligned}
[\![r]\!]^{\mathtt{op}}_L &= \begin{cases} n \text{ if } \mathcal{A}(L,r) = n\\ r \text{ otherwise}\end{cases}\\
[\![r_1 + r_2]\!]^{\mathtt{op}}_L &= \begin{cases} n & \text{if } \mathcal{A}(L,r_i) = n_i\\ & \quad \text{and } n = n_1 + n_2\\ r_2 & \text{if } \mathcal{A}(L,r_1) = 0\\ r_1 & \text{if } \mathcal{A}(L,r_2) = 0\\ n_1 + r_2 & \text{if } \mathcal{A}(L,r_1) = n_1\\ n_2 + r_1 & \text{if } \mathcal{A}(L,r_2) = n_2\\ r_1 + r_2 & \text{in any other cases}\end{cases}\\
[\![r_1 \lhd r_2]\!]^{\mathtt{cmp}}_L &= \begin{cases} \mathsf{true} & \text{if } \mathcal{A}(L,r_i) = n_i \text{ and } n_1 \lhd n_2\\ \mathsf{false} & \text{if } \mathcal{A}(L,r) = n_i \text{ and } \neg(n_1 \lhd n_2)\\ r_1 \lhd r_2 & \text{otherwise}\end{cases}
\end{aligned}
$$

Fig. 2.11. Constant Propagation

After applying $\mathsf{elim}_{\Rightarrow}$ (i.e., moving hypothesis to the context), and rewriting equalities from the context in the goal, one is left to prove closed equalities of the form $n = n'$ (i.e., n, n' are constants and do not contain variables). If the assertions are correct, then the certificate is obtained by applying reflexivity of equality (an instance of the ring rule).

Certificate translation

Suppose we have a certified function f with declaration $\{\vec{r_g};\ \varphi;\ G;\ \psi;\ \lambda;\ \vec{\Lambda}\}$. After applying constant propagation we get a function $\overline{f}$ and we are interested on building its corresponding certificates $\overline{\lambda}$ and $\overline{\vec{\Lambda}}$.

To build $\overline{\vec{\Lambda}}$ (the set proof obligations generated by the intermediate assertions), for each instruction of the form $\overline{f}[L] = (\varphi \wedge \mathrm{RES}_{\mathcal{A}}(L),\ \mathsf{ins})$ we have to find a proof for $\vdash \varphi \wedge \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$. To this end, we rely on an auxiliary function T^{ins}_L of type

$$T^{\mathsf{ins}}_L\ : \mathcal{C}(\vdash \mathsf{wp}^{\mathtt{id}}_f(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L))$$

for every label L, and on the function T_λ defined in Section 2.3.1 to construct a certificate for $\vdash \varphi \wedge \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathsf{id}}_{\overline{f}}(L)$. Furthermore, for every local substitution of op by $[\![\mathsf{op}]\!]^{\mathsf{op}}_L$ performed by the optimization, we require a certificate $T_{\mathsf{op}}(\mathsf{op}, L)$ for $\vdash \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \langle \mathsf{op} \rangle = \langle [\![\mathsf{op}]\!]^{\mathsf{op}}_L \rangle$.

The certificate T^{ins}_L represents the fact that under the hypothesis that the result of the analysis is correct, if a program state satisfies $\mathsf{wp}^{\mathsf{id}}_f(L)$ then it will also satisfy $\mathsf{wp}^{\mathsf{id}}_{\overline{f}}(L)$. The definition of the constructor T^{ins}_L is detailed in Fig. 2.12 for the assignment case. In the figure, the auxiliary function T_L of type $\mathcal{C}(\vdash \mathsf{wp}_f(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}_{\overline{f}}(L))$ is defined equal to T^{ins}_L when $f[L]$ does not contain an assertion. Otherwise, T_L has type $\mathcal{C}(\vdash \phi \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \phi \wedge \mathrm{RES}_{\mathcal{A}}(L))$ for some ϕ and, thus, it is trivially defined.

Let

$$
\begin{aligned}
&f[L] = r := \mathsf{op},\ L' \\
&\Gamma = [\phi_1; \mathrm{RES}_{\mathcal{A}}(L)] \\
&\phi_1 = \mathsf{wp}_f(L')[{}^{\mathsf{op}}/_{r_d}] \\
&\phi_2 = \mathrm{RES}_{\mathcal{A}}(L')[{}^{\mathsf{op}}/_{r_d}] \\
&\phi_3 = \mathsf{wp}_{\overline{f}}(L')[{}^{\mathsf{op}}/_{r_d}]
\end{aligned}
$$

in:

$$
\begin{aligned}
&p_1{:=}T_L(L') : \vdash \mathsf{wp}_f(L') \Rightarrow \mathrm{RES}_{\mathcal{A}}(L') \Rightarrow \mathsf{wp}_{\overline{f}}(L') \\
&p_2{:=}\mathsf{weak}\ \Gamma(\mathsf{subst}\ r_d\ \langle \mathsf{op} \rangle (p_1)) : \Gamma \vdash \phi_1 \Rightarrow \phi_2 \Rightarrow \phi_3 \\
&p_3{:=}\mathsf{axiom}(\phi_1) : \Gamma \vdash \phi_1 \\
&p_4{:=}\mathsf{weak}\ \Gamma(\vec{\Lambda}_{\mathcal{A}}(L))\Gamma \vdash \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \phi_2 \\
&p_5{:=}\mathsf{axiom}(\mathrm{RES}_{\mathcal{A}}(L)) : \Gamma \vdash \mathrm{RES}_{\mathcal{A}}(L) \\
&p_6{:=}\mathsf{elim}_{\Rightarrow}(p_5, p_4) : \Gamma \vdash \phi_2 \\
&p_7{:=}\mathsf{elim}_{\Rightarrow}(p_6, \mathsf{elim}_{\Rightarrow}(p_3, p_2)) : \Gamma \vdash \phi_3 \\
&p_8{:=}\mathsf{weak}\ \Gamma(T_{\mathsf{op}}(L, \mathsf{op})) : \Gamma \vdash \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \langle \mathsf{op} \rangle = \mathsf{interp}[\![\mathsf{op}]\!]^{\mathsf{op}}_L \\
&p_9{:=}\mathsf{elim}_{\Rightarrow}(p_5, p_8) : \Gamma \vdash \langle \mathsf{op} \rangle = \mathsf{interp}[\![\mathsf{op}]\!]^{\mathsf{op}}_L \\
&p_{10}{:=}\mathsf{elim}_{=}(p_9, p_7) : \Gamma \vdash \mathsf{wp}_{\overline{f}}(L')[{}^{\langle [\![\mathsf{op}]\!]^{\mathsf{op}}_L \rangle}/_{r_d}] \\
&p_{11}{:=}\mathsf{intro}_{\Rightarrow}(\mathsf{intro}_{\Rightarrow}(p_{10})) : \vdash \mathsf{wp}^{\mathsf{id}}_f(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathsf{id}}_{\overline{f}}(L)
\end{aligned}
$$

Fig. 2.12. Definition of T^{ins}_L for assignment case

Example 2.9. Consider as an example the following program transformation:

$$
\begin{array}{ll}
L_1 : \{r_2 \geq 1\} & L_1 : \{r_2 \geq 1\} \\
L_2 : r_1 := 1 & L_2 : r_1 := 1 \\
L_3 : \{r_2 \geq r_1 \wedge r_1 \geq 0\} & L_3 : \{r_2 \geq r_1 \wedge r_1 \geq 0\} \\
L_4 : r_3 := r_1 \quad \longrightarrow & L_4 : r_3 := 1 \\
L_5 : \{r_2 \geq r_3 \wedge r_3 = r_1 \wedge r_1 \geq 0\} & L_5 : \{r_2 \geq r_3 \wedge r_3 = r_1 \wedge r_1 \geq 0\} \\
L_6 : r_2 := r_2 + r_3 & L_6 : r_2 := r_2 + 1 \\
L_7 : \mathsf{nop},\ L_3 & L_7 : \mathsf{nop},\ L_3
\end{array}
$$

We have originally a proof obligation generated at L_3:

$$r_2 \geq r_1 \wedge r_1 \geq 0 \Rightarrow r_2 \geq r_1 \wedge r_1 = r_1 \wedge r_1 \geq 0$$

and a proof obligation at L_5:

$$r_2 \geq r_3 \wedge r_3 = r_1 \wedge r_1 \geq 0 \Rightarrow r_2 + r_3 \geq r_1 \wedge r_1 \geq 0 \ .$$

The first proof obligation becomes $r_2 \geq r_1 \wedge r_1 \geq 0 \Rightarrow r_2 \geq 1 \wedge 1 = r_1 \wedge r_1 \geq 0$ after the program transformation. Clearly, in order to obtain a proof of it, it is necessary to introduce the condition $r_1 = 1$ in the antecedent. This motivates the need for the hypothesis about the result of the analysis, in this case $r_1 = 1$.

However, this introduction is not always needed. For example, the second condition becomes $r_2 \geq r_3 \wedge r_3 = r_1 \wedge r_1 \geq 0 \Rightarrow r_2 + 1 \geq r_1 \wedge r_1 \geq 0$ in the optimized code, and the assertion $r_1 = 1$ is not necessary at L_5 in order to prove the verification condition (unless it is also introduced at L_3).

2.3.3 Loop Induction Variable Strength Reduction

Description

Loop induction strength reduction aims at reducing the number of multiplication operations inside a loop, which are commonly more costly than addition operations. An *induction register* is a register that is incremented (or decremented) in each iteration of the loop by a fixed constant value. An induction register is defined in the loop by an instruction of the form $r_i := r_i + c$, where c is a constant value. A *derived induction register* is a register that is assigned in each iteration of the loop the value of a linear function on the induction register. A derived induction register is defined in the loop by an instruction of the form $r_d := b * r_i$, where b is a constant value. The optimization consists of replacing any instruction updating a derived induction register r_d, made in terms of the basic induction register r_i, by an increment made only in terms of the previous value of r_d. For example, in

$$\begin{array}{rl} L_{oop} : & r_i := r_i + c \\ & r_d := b * r_i \\ & \mathsf{nop},\ L_{oop} \end{array}$$

r_i is a basic induction register with an increment of c, and r_d is an induction register with coefficient b derived from r_i. The optimization replaces the assignment $r_d := b * r_i$ by the less costly assignment $r_d := r_d + b * c$ and introduces the initialization $r_d := b * r_i$ just before the head of the loop. In addition to reducing the cost of the instruction, the live range of the register r_i is reduced, enabling further optimizations.

In the sequel, we follow the simplifying assumption that the loop body contains a single assignment for each register r_i and r_d. We assume also that the compiler has the ability to detect loops and returns a set of labels $\{L_1, \ldots, L_n\}$

determining the loop body, from which the header label L_H is the unique entry point.

Let $\{\vec{x};\ \varphi;\ G;\ \psi;\ \lambda;\ \vec{\Lambda}\}$ be the declaration for function f. Strength reduction proceeds in two steps. In the first step, an analysis detects inside the loop an induction register r_i and a derived induction register r_d. More precisely, the analysis takes as input a set of labels $\{L_1, \ldots, L_n\}$ (we assume that this set of labels corresponds to the output of a loop analysis, and that the header label is L_H) and provides the following information: an induction register r_i and the label L_i in which it is updated, a derived induction register r_d and the label L_d in which its definition appears, a fresh register name r'_d, two new labels L''_i and L''_H not in the domain of G_f and two constant values b, c that correspond to the coefficient of r_d and the increment of r_i, respectively. The first transformation step consists in introducing two assignments to a fresh register r'_d, one immediately before the loop header and the other one immediately after the assignment $f[L_i] = r_i := r_i + c,\ L'_i$. The output function at this transformation step is named f' and is defined in Figure 2.13. The motivation for this transformation phase is to ensure the invariance of the condition $r'_d = b * r_i$ in the loop body. In the figure, labels L', L'_H, L'_i, and L'_d are the successor labels for the instructions at L, L_H, L_i and L_d, respectively. To ensure that the instruction at L''_H is always executed before entering the loop, we update the labels outside the loop replacing L_H with L''_H.

$$
\begin{array}{lll}
f'[L''_H] & = r'_d := b * r_i,\ L_H & \\
f'[L_H] & = f[L_H] & \\
f'[L_i] & = r_i := r_i + c,\ L''_i & \\
f'[L''_i] & = r'_d := r'_d + b * c,\ L'_i & \\
f'[L] & = \begin{cases} f[L] & \text{if } L \text{ is any other label inside the loop} \\ f[L][^{L''_H}/_{L_H}] & \text{if } L \text{ is a label outside the loop} \end{cases}
\end{array}
$$

Fig. 2.13. Loop Induction: First Transformation Step

In a second step, an analysis determines the invariance of the condition $r'_d = b * r_i$ and consequently an optimization replaces the assignment to the derived induction register $r_d := b * r_i$ by the less costly assignment $r_d := r'_d$. We define the optimized function $\overline{f}$ as $\overline{f}[L_d] = r_d := r'_d,\ L'_d$ and $\overline{f}[L] = f'[L]$ for every $L \neq L_d$. In addition, every annotation inside the loop body is augmented with the condition $r'_d = b * r_i$. The annotations for labels outside the loop are not modified.

Certifying analyzer

Since the first transformation step just adds an assignment to a fresh register, only the analysis of the second step must be certified. To annotate f' with the

result of the analysis, we define $\mathrm{RES}_{\mathcal{A}}(L)$ as $r'_d = b * r_i$ if L is in $\{L_1, \ldots, L_n\}$, as $r'_d = b*(r_i - c)$ for $L = L''_i$, and $\mathrm{RES}_{\mathcal{A}}(L) \equiv \mathsf{true}$ in any other case (i.e., when L is a label outside the loop). Then, we need to create a certificate that the analysis is correct. The definition of $f'_{\mathcal{A}}$ is given by $f'_{\mathcal{A}}[L] = (\mathrm{RES}_{\mathcal{A}}(L),\ \mathsf{ins})$, where $f[L] = \mathsf{ins}$ or $f[L] = (\phi,\ \mathsf{ins})$. Since $\mathrm{RES}_{\mathcal{A}}(L)$ refers only to registers r'_d and r_i, the only interesting proof obligations are those corresponding to program labels L''_H, L_i, and L''_i.

Certificate translation

Certificate translation from a certificate for f into a certificate for $\overline{f}$ is also performed in two steps. In the first one, we build a certificate for f' from a certificate for f. Then we build a certificate for $\overline{f}$ from the certificate for f'.

The first translation step is trivial due to preservation of proof obligations. More precisely, we can show that since r'_d is a fresh register and, hence, does not appear in the code nor in assertions, the introduction of assignments targeting register r'_d does not affect the computation of the function $\mathsf{wp}^{\mathtt{id}}_f$. Formally, we can prove by the induction principle induced by the definition of well-annotated programs, that for every $L \notin \{L''_H, L''_i\}$, $\mathsf{wp}^{\mathtt{id}}_{f'}(L) = \mathsf{wp}^{\mathtt{id}}_f(L)$. Then, since no proof obligations are introduced at L''_H or L''_i, the set of proof obligation $\{\phi \Rightarrow \mathsf{wp}^{\mathtt{id}}_{f'}(L) \mid f'[L] = (\phi,\ \mathsf{ins})\}$ corresponds to the original set of proof obligations $\{\phi \Rightarrow \mathsf{wp}^{\mathtt{id}}_f(L) \mid f[L] = (\phi,\ \mathsf{ins})\}$. Therefore, the original certificate can be reused without modifications.

Certificate translation for the transformation from f' to $\overline{f}$ proceeds as with constant propagation or any certificate translation that requires a certifying analyzer. That is, we need to give explicitly a proof of $\mathsf{wp}_{\overline{f}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$ for each instruction of the form $\overline{f}[L] = (\varphi \wedge \mathrm{RES}_{\mathcal{A}}(L),\ \mathsf{ins})$. We can avoid repeating the main process since it is the same that was specified in the previous section (constant propagation). However it remains to define T^{ins}_L. Intuitively, the function T^{ins}_L expresses a correspondance between the weakest precondition function applied inside the loop of functions f' and $\overline{f}$, provided the hypothesis $r'_d = b * r_i$ is valid.

$$T^{\mathsf{ins}}_L\colon \mathcal{C}(\vdash \mathsf{wp}^{\mathtt{id}}_{f'}(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L))$$

This function T^{ins}_L is constructed using the induction principle attached to the definition of well-annotated programs, and then this result is merged with the original certificate and the certificate of the analysis, to produce a certificate for $\overline{f}$. In Figure 2.14, we show the definition of T^{ins}_L for the case $L = L_d$.

Example 2.10. Consider the following program, where $n \in \mathbb{N}$:

Let $\Gamma = [\mathsf{wp}^{\mathtt{id}}_{f'}(L_d), \mathrm{RES}_{\mathcal{A}}(L_d)]$ in

$$
\begin{aligned}
p_1 &:= T_L(L'_d) : \vdash \mathsf{wp}_{f'}(L'_d) \Rightarrow \mathsf{wp}_{f'_{\mathcal{A}}}(L'_d) \Rightarrow \mathsf{wp}_{\overline{f}}(L'_d)\\
p_2 &:= \mathsf{subst}\ r_d\ b{*}r_i(p_1) : \vdash \mathsf{wp}^{\mathtt{id}}_{f'}(L_d) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{f'_{\mathcal{A}}}(L_d) \Rightarrow \mathsf{wp}_{\overline{f}}(L'_d)[{}^{b*r_i}/_{r_d}]\\
p_3 &:= \mathsf{axiom}(\mathsf{wp}^{\mathtt{id}}_{f'}(L_d)) : \ \Gamma \vdash \mathsf{wp}^{\mathtt{id}}_{f'}(L_d)\\
p_4 &:= \mathsf{axiom}(\mathrm{RES}_{\mathcal{A}}(L_d)) : \ \Gamma \vdash \mathrm{RES}_{\mathcal{A}}(L_d)\\
p_5 &:= \mathsf{weak}\ \Gamma(\vec{\Lambda}_{\mathcal{A}}(L_d)) : \ \Gamma \vdash \mathrm{RES}_{\mathcal{A}}(L_d) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{f'_{\mathcal{A}}}(L_d)\\
p_6 &:= \mathsf{elim}_{\Rightarrow}(p_4, p_5) : \ \Gamma \vdash \mathsf{wp}^{\mathtt{id}}_{f'_{\mathcal{A}}}(L_d)\\
p_7 &:= \mathsf{elim}_{\Rightarrow}(p_3, \mathsf{weak}\ \Gamma(p_2)) : \ \Gamma \vdash \mathsf{wp}^{\mathtt{id}}_{f'_{\mathcal{A}}}(L_d) \Rightarrow \mathsf{wp}_{\overline{f}}(L'_d)[{}^{b*r_i}/_{r_d}]\\
p_8 &:= \mathsf{elim}_{\Rightarrow}(p_6, p_7) : \ \Gamma \vdash \mathsf{wp}_{\overline{f}}(L'_d)[{}^{b*r_i}/_{r_d}]\\
p_9 &:= \mathsf{elim}_{=}(p_4, p_8) : \ \Gamma \vdash \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L_d)\\
T^{\mathsf{ins}}_L(L_d) &:= \mathsf{intro}_{\Rightarrow}(\mathsf{intro}_{\Rightarrow}(p_9)) : \vdash \mathsf{wp}^{\mathtt{id}}_{f'}(L_d) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L_d) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L_d)
\end{aligned}
$$

Fig. 2.14. Definition of T^{ins}_L for the case $L = L_d$

$$
\begin{aligned}
& r := 0\\
& r_i := 0\\
L_{oop} : & \{r = b * r_i \wedge r_i \leq n\}\\
& r_i \geq n\ ?\ L_{out}\\
& r_i := r_i + 1\\
& r_d := b * r_i\\
& r := r_d\\
& \mathsf{nop},\ L_{oop}\\
L_{out} : & \{r = b * n\}
\end{aligned}
$$

One can apply the optimization of this section to reduce the strength of the instruction updating the derived induction register r_d. Focusing on the proof obligation corresponding to the preservation of the loop invariant, we can extract the interesting fragment

$$r = b * r_i \wedge r_i \leq n \wedge r_i < n \Rightarrow b * (r_i + 1) = b * (r_i + 1)\ .$$

If the program is transformed as follows (strength reduction)

$$
\begin{aligned}
& r := 0\\
& r_i := 0\\
& r'_d := 0\\
L_{oop} : & \{r = b * r_i \wedge r_i \leq n\}\\
& r_i \geq n\ ?\ L_{out}\\
& r_i := r_i + 1\\
& r'_d := r'_d + b\\
& r := r'_d\\
& \mathsf{nop},\ L_{oop}\\
L_{out} : & \{r = b * n\}
\end{aligned}
$$

the fragment of the proof obligation becomes

$$r = b * r_i \wedge r_i \leq n \wedge r_i < n \Rightarrow r'_d + b = b * (r_i + 1) \ .$$

The condition $r'_d = b * r_i$ is missing in the antecedent, and adding it in is mandatory to verify the code. At the same time, this extension of the invariant must be verified, a task that corresponds to the proof obligations automatically generated by the certifying analyzer.

2.3.4 Common Subexpression Elimination

Description

Common subexpression elimination (CSE) aims at reducing the number of duplicated computations by reusing previously defined and still available non-trivial expressions: if the same expression is computed at two different program points, CSE eliminates one of the computations, by replacing the second operation by an access to the register containing the result of the first evaluation.

CSE is similar to constant propagation, in the sense that the transformation is triggered by conditions represented by an equality between a register and an expression. In constant propagation, this expression corresponds to a constant value, whereas in CSE it may be a more complex expression (commonly involving arithmetic operators). Therefore, the principle behind certificate translation for CSE is very similar to the one for certificate translation for CP. In the following example

$$\begin{array}{l} r_1 := r_2 + r_3 \\ r_4 := r_2 + r_3 \\ r_1 := r_1 + 1 \\ r_5 := r_2 + r_3 \end{array} \quad \longrightarrow \quad \begin{array}{l} r_1 := r_2 + r_3 \\ r_4 := r_1 \\ r_1 := r_1 + 1 \\ r_5 := r_4 \end{array}$$

the assignment updating the register r_4 could be optimized, but after the second assignment over r_1, the availability of the expression $r_2 + r_3$ through r_1 is lost, and then the assignment to r_5 has to be optimized with r_4.

We begin by discussing the certifying analyzer for CSE. The function $\mathcal{A}$ with type $\mathcal{PP} \times \mathcal{R} \rightarrow \mathsf{op}_\perp$ associates to a label L and a register r an RTL expression. Similarly to constant propagation, the assertion $\mathrm{RES}_\mathcal{A}$ is defined as:

$$\mathrm{RES}_\mathcal{A}(L) \equiv \bigwedge_{\mathcal{A}(L,r) \neq \perp} r = \langle \mathcal{A}(L,r) \rangle$$

where $\langle \mathcal{A}(L,r) \rangle$ is the interpretation into the language of assertions of the information returned by the analysis.

The certifying analyzer generates automatically a certificate for $f_\mathcal{A}$, where for every label L, the corresponding annotation will be $\mathrm{RES}_\mathcal{A}(L)$.

We briefly describe the transformation for CSE. For each $f[L]$ of the form $r := e$, if there is a register r' such that $r' \neq r$ and $\mathcal{A}(L, r') = e$, we replace the instruction $f[L]$ by $r := r'$.

Finally, certificate translation proceeds exactly as with constant propagation, with the definition of two mutually recursive transfer functions using the induction principle attached to the definition of $\mathsf{reachAnnot}_f$:

$$T_L^{\mathsf{ins}}\colon \forall L,\ \mathcal{C}(\vdash \mathsf{wp}_f^{\mathtt{id}}(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}_{\overline{f}}^{\mathtt{id}}(L))$$
$$T_L\ \colon \forall L,\ \mathcal{C}(\vdash \mathsf{wp}_f(L) \Rightarrow \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}_{\overline{f}}(L))$$

Then, these two functions are used to merge the original certificate with the certificate of the analysis to build a certificate for $\overline{f}$.

Example 2.11. Consider the previous example with intermediate assertions:

$$\begin{array}{l}
\{0 \leq r_2 \wedge 0 \leq r_3\} \\
r_1 := r_2 + r_3 \\
\{0 \leq r_2 \wedge 0 \leq r_3\} \\
r_4 := r_2 + r_3 \\
\{0 \leq r_4 \wedge 0 \leq r_2 \wedge 0 \leq r_3\} \\
r_1 := r_1 + 1 \\
r_5 := r_2 + r_3 \\
\{0 \leq r_5\}
\end{array}
\quad \rightarrow \quad
\begin{array}{l}
\{0 \leq r_2 \wedge 0 \leq r_3\} \\
r_1 := r_2 + r_3 \\
\{0 \leq r_2 \wedge 0 \leq r_3\} \\
r_4 := r_1 \\
\{0 \leq r_4 \wedge 0 \leq r_2 \wedge 0 \leq r_3\} \\
r_1 := r_1 + 1 \\
r_5 := r_4 \\
\{0 \leq r_5\}
\end{array}$$

If the instruction $r_4 := r_2 + r_3$ is replaced with $r_4 := r_1$, the lack of information about register r_1 on assertions prevents proving the condition $0 \leq r_4$ of the third assertion. Then, we need to propagate $r_1 = r_2 + r_3$ to the second assertion. Finally, although there is enough information about register r_4 on the second assertion to prove that $0 \leq r_5$ is valid after the assignment, the original proof is done in terms of r_2 and r_3. Hence, the need for a general and automatic technique forces us to introduce the condition $r_4 = r_2 + r_3$.

The $f_{\mathcal{A}}$ and $\overline{f}$ functions are respectively:

$$\begin{array}{ll}
\{\mathsf{true}\} & \{0 \leq r_2 \wedge 0 \leq r_3\} \\
r_1 := r_2 + r_3 & r_1 := r_2 + r_3 \\
\{r_1 = r_2 + r_3\} & \{0 \leq r_2 \wedge 0 \leq r_3 \wedge r_1 = r_2 + r_3\} \\
r_4 := r_2 + r_3 & r_4 := r_1 \\
\{r_4 = r_2 + r_3\} & \{0 \leq r_4 \wedge 0 \leq r_2 \wedge 0 \leq r_3 \wedge r_4 = r_2 + r_3\} \\
r_1 := r_1 + 1 & r_1 := r_1 + 1 \\
r_5 := r_2 + r_3 & r_5 := r_4 \\
\{\mathsf{true}\} & \{0 \leq r_5\}
\end{array}$$

2.3.5 Copy Propagation

Description

Copy propagation aims at reducing the live range of variables defined by move operations, simplifying register allocation at the code generation phase.

It is intended to remove the number of auxiliary register copies that may be introduced by other optimizations.

Copy propagation searches for occurrences of instructions of the form $r_1 := r_2$, and replaces every occurrence of r_1 by r_2 (along its successors' path) until either r_1 or r_2 are modified. The original assignment $r_1 := r_2$ can be moved forward (down) in the code until the condition $r_1 = r_2$ gets invalidated.

Example 2.12. Consider the original program

$$\begin{array}{l} r_1 := r_2 \\ S(r_1, r_2) \\ r_3 := r_1 + r_2 \\ r_2 := r_1 \ \mathsf{op} \ r_2 \\ T(r_1, r_2) \end{array}$$

where S and T represent sequences of instructions and neither r_1 or r_2 are modified in S. Copy propagation of the variable r_2 results in the following program:

$$\begin{array}{l} r_1 := r_2 \\ S(r_2, r_2) \\ r_3 := r_2 + r_2 \\ r_2 := r_2 \ \mathsf{op} \ r_2 \\ T(r_1, r_2) \end{array}$$

The transformation is advantageous in terms on availability of register r_1, since its live range can be reduced considerably, and in consequence one may move the first assignment forward to get:

$$\begin{array}{l} S(r_2, r_2) \\ r_3 := r_2 + r_2 \\ r_1 := r_2 \\ r_2 := r_2 \ \mathsf{op} \ r_2 \\ T(r_1, r_2) \end{array}$$

Certificate translation

As with constant propagation, common subexpression elimination, and many other optimizations, the translation of the certificate can be made by using a certifying analyzer. First, a special purpose function f_A is generated, fully annotated with the results of the dataflow analysis (e.g., $r_1 = r_2$), and a new certificate is automatically generated validating these auxiliary assertions. We formallize the result of the analysis as a function $\mathcal{A} : \mathcal{PP} \times \mathcal{R} \to \mathcal{R}_\perp$, such that, for any $r_1, r_2 \in \mathcal{R}$, $\mathcal{A}(L, r_1) = r_2$ only if r_1 holds a copy of the value at r_2 at program point L. The result of the analysis $\mathcal{A}$ is used to define

$$\mathrm{RES}_{\mathcal{A}}(L) \equiv \bigwedge_{r \in \{r \mid \mathcal{A}(L,r) \neq \perp\}} r = \mathcal{A}(L, r)$$

and to generate a certificate for the analysis, defining for each reachable label L the certificate

$$\vdash \mathrm{RES}_{\mathcal{A}}(L) \Rightarrow \mathsf{wp}^{\mathsf{id}}_{f,\mathcal{A}}(L)$$

that is later integrated with the certificate of the original program. The function T^{ins}_L is defined by case analysis. The proof for T^{ins}_L does not represent any further difficulty with respect to that of constant propagation or common subexpression elimination.

2.3.6 Dead Register Elimination

Description

Dead register elimination aims at removing assignments to registers that will not be needed in the future. As mentioned in the introduction, we propose a transformation that removes dead assignments while simultaneously modifying the assertions. Regarding certificate translation, this program transformation is atypical since it does not follow the scheme consisting on a certifying analyzer and a definition of the function T^{ins}_L.

Intuitively, a register is live at some execution point if it stores a value that interferes with the subsequent execution steps. The classical definition is intentional and overapproximating: a register r is live at label L if r is read at label L or there is a path from L that reaches a label L' where r is read and does not go through an instruction that defines r (including L, but not L'). A register r is read at label L if it appears as a parameter of a function call, in a conditional jump, in a `return` instruction, or on the right side of an assignment to a live register r'.

In the following, we represent with the function $\mathcal{L}$ the result of the analysis and write $\mathcal{L}(L, r) = \top$ when a register is live at L.

A live range of a register r is a set of sequences of consecutive input or output edges such that r is live.

The problem with certificate translation for removal of dead registers is that intermediate assertions can specify conditions over local variables that will never be used, and hence are dead in the program. Arguably, we may assume that original assertions only refer to live variables. However, some optimizations can reduce the live range of variables and, therefore, the occurrence of some registers may become redundant after previous optimizations steps. For example in the following optimization

$$\begin{array}{ll} \{\mathsf{true}\} & \{\mathsf{true}\} \\ r_1 := 1 & r_1 := 1 \\ \{r_1 = 1\} \quad \longrightarrow & \{r_1 = 1\} \\ r_2 := r_1 & r_2 := 1 \\ \{r_2 = 1\} & \{r_2 = 1\} \end{array}$$

the register r_1 becomes dead, but the assignment $r_1 := 1$ cannot be removed since we have to ensure that $r_1 = 1$ is a valid intermediate assertion.

In order to deal with assertions, we extend the definition of liveness.

Definition 2.13 (Live in assertions). *A register r is live in an assertion at label L, denoted by $\mathcal{L}(L, r) = \top_\phi$, if it is not live at label L but there is a path from L that reaches a label L' such that r appears in an assertion at L' or where r is used to define a register which is live in an assertion at label L'.*

By abuse of notation, we write $\mathcal{L}(L, r) = \bot$ if r is dead in the code and in assertions.

In order to deal with registers that are dead in the code but live in assertions, we rely on the introduction of ghost variables. Ghost variables are expressions in our language of assertions (we assume that the set of ghost variables names and the set of register names $\mathcal{R}$ are disjoint). We introduce, as part of the set of RTL instructions, *ghost assignments*, i.e., instructions of the form $\mathsf{set}\ \hat{v} := \mathsf{op},\ L$, where $\hat{v}$ is a ghost variable. Ghost assignments do not affect the semantics of RTL, but they affect the calculus of wp in the same way as normal assignments. At the end of this section we discuss the soundness of the verification infrastructure in presence of ghost variables.

In the following, if σ is a function mapping expressions to registers, we denote $\phi\sigma$ the result of substituting every free register r in ϕ by σr.

We propose a transformation, defined by the following equations:

$$\begin{aligned} \mathsf{ghost}_L((\varphi,\ \mathsf{ins})) &= (\varphi\sigma_L,\ \mathsf{ghost}_L^{\mathtt{id}}(\mathsf{ins})) \\ \mathsf{ghost}_L(\mathsf{ins}) &= \mathsf{ghost}_L^{\mathtt{id}}(\mathsf{ins}) \end{aligned}$$

where the substitution σ_L maps r to $\hat{r}$ if $\mathcal{L}(L, r) = \top_\phi$ and $\mathsf{dead}_c(L, L') = \{r \mid \mathcal{L}(L, r) = \top \wedge \mathcal{L}(L', r) = \top_\phi\}$ and the transformation function $\mathsf{ghost}_L^{\mathtt{id}}(\mathsf{ins})$ is defined in Figure 2.15. We use $\mathsf{set}\ \hat{\vec{r}} := \vec{r}$ as syntactic sugar for a sequence of assignments $\mathsf{set}\ \hat{r_i} := r_i$ where for each register r_i in the sequence $\vec{r}$, $\hat{r_i}$ in $\hat{\vec{r}}$ is its corresponding ghost variable. Each instruction of f is transformed into a sequence of instructions for $\overline{f}$. Intuitively, it introduces for every instruction ins (with successor L') at label L, a ghost assignment $\mathsf{set}\ \hat{r} := r,\ L'$ immediately after L (at a new label L'') if the register r is live at L but not live at the immediate successor L' of L. In addition, every instruction of the form $r_d := \mathsf{op}$ is removed if the register r_d is not live at L.

Example 2.14. Taking as input the short piece of code from the introduction, the result of the transformation is:

$$\begin{array}{l} \{\mathsf{true}\} \\ \mathsf{set}\ \hat{r_1} := 1 \\ \{\hat{r_1} = 1\} \\ r_2 := 1 \\ \{r_2 = 1\} \end{array}$$

Certificate translation

Certificate translation for dead register elimination falls in the IPO category, i.e., the certificate of the optimized program is an instance of the certificate of the source program. This is shown by proving that ghost variable introduction preserves annotations up to substitution of dead variables.

Lemma 2.15. *After applying ghost variable introduction, the transformed function $\overline{f}$ satisfies the following property:*

$$\forall L, \mathsf{wp}_{\overline{f}}(L) = \mathsf{wp}_f(L)\sigma_L$$

where σ is the substitution defined above.

Proof. *The proof is by the induction principle attached to the definition of* reachAnnot. *We only consider some representative cases:*

— *Case $f[L] = (\varphi,\ \mathsf{ins})$. In this case $\mathsf{wp}_{\overline{f}}(L)$ is equal to $\varphi\sigma_L$ by definition of* wp *and ghost variable introduction. But, since φ is equal to $\mathsf{wp}_f(L)$, the lemma is satisfied.*

— *Case $f[L] = \mathsf{nop},\ L'$. By definition of* wp*, we have that $\mathsf{wp}_{\overline{f}}(L)$ is equal to $\mathsf{wp}_{\overline{f}}(L')$, which in turn is equal to $\mathsf{wp}_f(L')\sigma_{L'}$ by inductive hypothesis. Since $\mathsf{wp}_f(L') = \mathsf{wp}_f(L)$, it remains to prove that $\sigma_{L'}$ is in fact equal to σ_L, but this is the case since the condition of liveness or liveness in assertion is the same for L and L' for any register. For example, if $\mathcal{L}(L', r) = \top$, that means that r is read at label L' or there is a path π that reaches label L'' such that r is never assigned. In the first case, we can propose $L \to L'$ as a path from L that reaches L' where r is read. And in the second case we can construct the desired path appending $L \to L'$ to π.*

— *Case $f[L] = r_d := \mathsf{op},\ L'$ and $\mathcal{L}(L', r_d) = \top_\phi$. By definition of* wp *and the transformation performed we have that $\mathsf{wp}_{\overline{f}}(L)$ is equal to $\mathsf{wp}_{\overline{f}}(L')[{}^{\mathsf{op}\sigma_L}/_{\hat{r_d}}]$, and by I.H., to $\mathsf{wp}_f(L')\sigma_{L'}[{}^{\mathsf{op}\sigma_L}/_{\hat{r_d}}]$. Now we have to see that*

$$[{}^{\mathsf{op}\sigma_L}/_{\hat{r_d}}] \circ \sigma_{L'} = \sigma_L \circ [{}^{\mathsf{op}}/_{r_d}]\ .$$

To prove this equation we proceed by case analysis. First, consider the application of the substitutions to register r_d, in this case we can see that both expressions are equivalent to $\mathsf{op}\sigma_L$. Now suppose that we apply them to a register $r \neq r_d$. In this case, if r is in $\mathsf{wp}_f(L')$ then it is live or live in assertion on label L', in which case, it will also be live or live in assertion, respectively, on label L. If r is live in assertion on label L, and r occurs free in $\mathsf{wp}_f(L')$, then it must also be the case that r is live in assertion on label L', because if it is not the case, and r occurs in $\mathsf{wp}_f(L')$, then r must be live on label L', which implies that is is also live on label L, and that is a contradiction.

A consequence of this lemma is that if the function f is certified, then it is possible to reuse the certificate of f to certify $\overline{f}$. More precisely, for

every label L s.t. $f[L]$ is of the form $(\phi_L,\ \mathsf{ins})$ we can obtain a proof of $\vdash \phi_L\sigma_L \Rightarrow \mathsf{wp}^{\mathtt{id}}_f(L)\sigma_L$ (i.e., of $\vdash \mathsf{wp}_{\overline{f}}(L) \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}(L)}$) by applying subst rule of Figure 2.5 to the original proof for $\vdash \phi_L \Rightarrow \mathsf{wp}^{\mathtt{id}}_f(L)$.

After ghost variable introduction has been applied, every register that occurs free in $\mathsf{wp}_{\overline{f}}(L)$ is live at L.

$$
\begin{array}{lll}
\mathsf{ghost}^{\mathtt{id}}_L(\mathsf{return}\ r) & = \mathsf{return}\ r & \\
\mathsf{ghost}^{\mathtt{id}}_L(r_d := f(\vec{r}),\ L') & = \left|\begin{array}{l} L\ \ : r_d := f(\vec{r}),\ L'' \\ L'' : \mathsf{set}\ \hat{\vec{t}} := \vec{t},\ L' \end{array}\right. & \\
\mathsf{ghost}^{\mathtt{id}}_L(\mathsf{nop},\ L') & = \mathsf{nop},\ L' & \\
\mathsf{ghost}^{\mathtt{id}}_L(\mathsf{cmp}\ ?\ L_1 : L_2) & = \left|\begin{array}{l} L\ \ : \mathsf{cmp}\ ?\ L_1' : L_2' \\ L_1' : \mathsf{set}\ \hat{\vec{t}}_1 := \vec{t}_1,\ L_1 \\ L_2' : \mathsf{set}\ \hat{\vec{t}}_2 := \vec{t}_2,\ L_2 \end{array}\right. & \\
\mathsf{ghost}^{\mathtt{id}}_L(r_d := \mathsf{op},\ L') & = \left\{\begin{array}{ll} \mathsf{nop},\ L' & \text{if } \mathcal{L}(L', r_d) = \bot \\ \mathsf{set}\ \hat{r}_d := \mathsf{op}\sigma_L,\ L' & \text{if } \mathcal{L}(L', r_d) = \top_\phi \\ \left|\begin{array}{l} L\ \ : r_d := \mathsf{op},\ L'' \\ L'' : \mathsf{set}\ \hat{\vec{t}} := \vec{t},\ L' \end{array}\right. & \text{if } \mathcal{L}(L', r_d) = \top \end{array}\right. &
\end{array}
$$

where $\vec{t}$, $\vec{t_1}$ and $\vec{t_2}$ stand respectively for variables in the sets $\mathsf{dead}_c(L, L')$, $\mathsf{dead}_c(L, L_1)$ and $\mathsf{dead}_c(L, L_2)$

Fig. 2.15. Ghost Variable Introduction-Dead Register Elimination

Soundness of Ghost Variable Introduction.

To consider the proposed transformation as an optimizing one, the execution environment must be capable of safely ignoring ghost assignments. Intuitively, the set of ghost variables do not affect the state of any other variable nor the control flow (that is because ghost variables do not appear on ordinary assignments or conditional jumps).xs If we define an equivalence relation between steps that only takes into consideration the coincidence on live variables, it should be clear that two executions starting from different but equivalent states always reach equivalent intermediate states. Stated in other words, the domain of ghost variables does not interfere with the domain of standard variables and, hence, it is safe to slice them out of the standard RTL programs.

Discussion.

We can avoid introducing ghost variables if we are able to remove dead variables from assertions. However, it is not trivial to determine whether a subassertion can be removed from an invariant. Consider for instance the following example:

$$
\begin{array}{rl}
 & y := 3 \\
 & x := y \\
L_{oop} : & \{\phi\} \\
 & b \ ?\ L_{out} \\
 & x := x + 1 \\
 & y := 3,\ L_{oop} \\
L_{out} : & \{x \geq 0\}
\end{array}
$$

where ϕ is $x \geq 3 \wedge y \geq 3$. Clearly, the assignment $y := 3$ within the loop and the subassertion $y \geq 3$ may be sliced out from the program. However, in other examples, it may be the case that the subassertion refers both to dead and live variables, and hence cannot be removed. Consider the example above but suppose now that ϕ is defined as $x \geq y \wedge y \geq 3$. In this case, the condition $y \geq 3$ appearing in the invariant is necessary to prove the invariance of $x \geq y$. Thus, ϕ may not be simplified.

An alternative approach, shown in Chapter 3, is to existentially quantify each assertion over its corresponding dead variables. Even though this is a simpler approach that does not require the existence of ghost variables in the verification framework, the result of transformation is a weaker program specification. Consider the following example and both alternative transformations (in that order):

$$
\begin{array}{ccc}
\{x = y\} & \{x = \hat{y}\} & \{\exists y.\ x = y\} \\
c & c & c \\
\{x = y\} & \{x = \hat{y}\} & \{\exists y.\ x = y\}
\end{array}
$$

where c is a sequence of instructions in which y does not occur, and y is considered a dead variable. On the left, in which ghost variable introduction has been applied, the final contract specifies that the value of x is the same after the execution of c. On the right, the relation between the original and final value of x is lost.

2.3.7 Unreachable Code Elimination

Description

Unreachable code elimination aims at removing instructions that are never executed.

The optimization described here simply identifies the connected graph of nodes that can be reached starting from the initial node pointed by L_{sp}, and removes the remaining nodes. The existence of unreachable program points may be originated, for instance, by application of other program optimizations.

We are not considering in this section the detection and removal of redundant conditional branches. Removing redundant conditional branches can be performed in a previous phase, in which the analysis is required to detect the validity of the branching condition. For example in

$$
\begin{array}{ll}
 & r_1 := r_2 \\
 & r_1 = r_2 \ ?\ L_t \\
 & [S] \\
L_t : & ...
\end{array}
\qquad\qquad
\begin{array}{ll}
 & r_1 := r_2 \\
 & \mathsf{nop},\ L_t \\
 & [S] \\
L_t : & ...
\end{array}
$$

the analysis may infer that the fact $r_1 = r_2$ is valid immediately before the conditional branch and, in addition, in order to translate the certificate, this condition must be proved correct. Next, as is common with several optimizations, the function T_L^{ins} is defined and the proof of the analysis is merged with the original proof to translate the certificate. Notice that the fragment of code $[S]$ is not removed since it may be reachable by some other instructions, this transformation is left to a different transformation: unreachable code elimination.

The selection of reachable nodes is a straightforward process and translating the certificate does not represent any difficulty.

Certificate translation

Our definition of VCgen is such that proof obligations are totally independent from unreachable nodes.

Its clear that for any label L reachable from L_{sp}, $\mathsf{wp}_f(L) = \mathsf{wp}_{f_{|R}}(L)$, where $f_{|R}$ is f with its graph code restricted to the set R of reachable labels. Notice that for the preservation of certificates we require the VCgen to consider only assertions on reachable labels, as is the case in this chapter. Otherwise, if the VCgen considers also annotations on unreachable labels, the set of proof obligations of the transformed program is a subset of the proof obligations of the initial program.

2.3.8 Register Allocation

Description

Register allocation is a code generation phase that intends to minimize register usage by storing in a single real machine register two or more temporary registers. That is possible, for instance, for those registers whose live ranges are disjoint (namely nonoverlapping registers). For simplicity, we abstract from the complexity of determining the mapping between pseudo-registers and final registers. We also refrain from considering cases in which the lack of available registers forces one to temporarily store them in a secondary memory (memory spills).

The result of the analysis is represented as a mapping σ with type $\mathcal{R} \mapsto \mathcal{R}$. The code transformation rewrites each instruction and assertion by applying the substitution σ given by the analysis, that maps temporary registers of the intermediate RTL language to a potentially shared register in the object language.

To ensure correctness of the analysis, we require that applying the transformation induced by this substitution σ is semantics preserving. More precisely, the following property is desired over the result of the analysis: for any registers r_1 and r_2, if $\sigma r_1 = \sigma r_2$ then the live ranges of r_1 and r_2 are disjoint. In addition, we require that there are no assignments to dead registers. This condition can be fulfilled by applying ghost variable introduction in a previous phase.

Certificate translation

If the substitution σ returned by the analysis is correct, i.e., two overlapping registers r_1 and r_2 are not mapped to the same register, then the proof obligations of the original and optimized program coincide up to variable renaming.

Lemma 2.16. *Suppose we have a function f, such that for any register r and any label L, $\mathcal{L}(L, r) \neq \top_\phi$ (all registers in assertions are live). Assume also that there are no assignments to dead variables. If $\overline{f}$ is the result of applying register allocation, then for any label L*

$$\mathsf{wp}_{\overline{f}}(L) = \mathsf{wp}_f(L)\sigma$$

where σ represents the mapping that joins pairs of nonoverlapping registers.

Proof. We prove the validity of the condition $\mathsf{wp}_{\overline{f}}(L) = \mathsf{wp}_f(L)\sigma$, using the induction principle induced by the definition of well-annotated program.

— base case $f[L] = (\varphi,\ \mathsf{ins})$
By definition of wp *and the register replacement,* $\mathsf{wp}_{\overline{f}}(L)$ *is equal to* $\varphi\sigma$*, which is equal to* $\mathsf{wp}_f(L)\sigma$

— base case $f[L] = \mathsf{return}\ r$
$\mathsf{wp}_{\overline{f}}(L)$ *is* $\mathsf{post}(f)[{}^{\sigma r}/_{\mathsf{res}}]$ *by definition, but it is clear that this is also equal to* $\mathsf{post}(f)[{}^{r}/_{\mathsf{res}}]\sigma$*. Which by definition is* $\mathsf{wp}_f(L)\sigma$*.*

— case $f[L] = \mathsf{nop},\ L'$
In this case $\mathsf{wp}_{\overline{f}}(L)$ *is equal to* $\mathsf{wp}_{\overline{f}}(L')$*, and by inductive hypothesis is equal to* $\mathsf{wp}_f(L')\sigma$*, and then to* $\mathsf{wp}_f(L)\sigma$*.*

— case $f[L] = r_d := g(\vec{r}),\ L'$
By definition of wp*, we have that* $\mathsf{wp}_{\overline{f}}(L)$ *is equal to*

$$\mathsf{pre}(f)[{}^{\sigma\vec{r}}/_{\vec{r}_g}] \wedge (\forall \mathsf{res}, \mathsf{post}(f)[{}^{\sigma\vec{r}}/_{\vec{r}^*_g}] \Rightarrow \mathsf{wp}_{\overline{f}}(L')[{}^{\mathsf{res}}/_{\sigma r_d}])$$

Since σ is the identity for variables occurring in $\mathsf{pre}(f)$ *or* $\mathsf{post}(f)$ *we have*

$$(\mathsf{pre}(f)[{}^{\vec{r}}/_{\vec{r}_g}])\sigma \wedge (\forall \mathsf{res}, (\mathsf{post}(f)[{}^{\vec{r}}/_{\vec{r}^*_g}])\sigma \Rightarrow \mathsf{wp}_{\overline{f}}(L')[{}^{\mathsf{res}}/_{\sigma r_d}])$$

and applying inductive hypothesis on $\mathsf{wp}_{\overline{f}}(L')$ *we get*

$$(\mathsf{pre}(f)[{}^{\vec{r}}/_{\vec{r}_g}])\sigma \wedge (\forall \mathsf{res}, (\mathsf{post}(f)[{}^{\vec{r}}/_{\vec{r}^*_g}])\sigma \Rightarrow \mathsf{wp}_f(L')\sigma[{}^{\mathsf{res}}/_{\sigma r_d}])$$

finally, since $\sigma\mathsf{res} = \mathsf{res}$*, it is clear that the last expression is equal to*

$$(\mathsf{pre}(f)[\vec{r}/\vec{r_g}])\sigma \wedge (\forall \mathsf{res}, (\mathsf{post}(f)[\vec{r}/\vec{r_g^*}])\sigma \Rightarrow (\mathsf{wp}_f(L')[^{\mathsf{res}}/_{r_d}])\sigma)$$

that is exactly $\mathsf{wp}_f(L)\sigma$

— *case* $f[L] = \mathsf{cmp}\ ?\ L_t : L_f$

In this case $\mathsf{wp}_{\overline{f}}(L)$ *is* $\mathsf{cmp}\sigma \Rightarrow \mathsf{wp}_{\overline{f}}(L_t) \wedge \neg\mathsf{cmp}\sigma \Rightarrow \mathsf{wp}_{\overline{f}}(L_f)$ *which by inductive hypothesis is equal to* $\mathsf{cmp}\sigma \Rightarrow \mathsf{wp}_f(L_t)\sigma \wedge \neg\mathsf{cmp}\sigma \Rightarrow \mathsf{wp}_f(L_f)\sigma$*. But this is in fact* $\mathsf{wp}_f(L)\sigma$ *by definition of* wp*.*

— *case* $f[L] = r_d := \mathsf{op},\ L'$

$\mathsf{wp}_{\overline{f}}(L)$ *is by definition of* wp *and the transformation,* $\mathsf{wp}_{\overline{f}}(L')[^{\mathsf{op}\sigma}/_{\sigma r_d}]$*. By inductive hypothesis, this is also equal to* $\mathsf{wp}_f(L')\sigma[^{\mathsf{op}\sigma}/_{\sigma r_d}]$*. If we show that for any variable* r *in* $\mathsf{wp}_f(L')$*,* $[^{\mathsf{op}\sigma}/_{\sigma r_d}](\sigma r)$ *is equal to* $\sigma([^{\mathsf{op}}/_{r_d}]r)$ *then* $\mathsf{wp}_f(L')\sigma[^{\mathsf{op}\sigma}/_{\sigma r_d}]$ *is equal to* $\mathsf{wp}_f(L')[^{\mathsf{op}}/_{r_d}]\sigma$*, and that is what we want to prove. To prove the equality between the two mappings, suppose first that* $r = r_d$*, in this case* $[^{\mathsf{op}\sigma}/_{\sigma r_d}](\sigma r)$ *is* $\mathsf{op}\sigma$ *and is clearly the same for* $\sigma([^{\mathsf{op}}/_{r_d}]r)$*. In case* $r \neq r_d$*,* σr *must be clearly different to* σr_d*, as* r_d *is live in* L' *and if* $r \in FV(\mathsf{wp}_f)(L)$ *then* r *is also live in* L'*. Under this assumption, both sides of the equation are equal to* σr*.*

Conclusion

As with dead register elimination, the proof transformation is just a renaming using the operator subst to the original certificates, as specified by the substitution function σ.

2.3.9 Function Inlining

Description

An immediate motivation of function inlining is reducing the overhead of the function call. However, its main purpose is to generate new optimization opportunities.

We do not consider here profitability issues, since it is not the purpose of this presentation. Indeed, even when the transformation reduces the call overhead and gives rise to new optimization opportunities, it is possibly undesirable since it may increase the code size and the number of registers in use. The analysis that follows is restricted to translating the certificate and does not focus on its profitability.

Suppose we have a function call $r_d := g(\vec{x}),\ L_{ret}$ as the instruction $f[L_{call}]$. The transformation consists of replacing this statement with an assignment to function g's formal parameters, followed by the body of the function g where every return instruction is replaced by a corresponding assignment to the register r_d. In this section, we assume for simplicity that the set of registers used by g is disjoint from those of the function where it will be inlined. Similarly, for notational convenience, we assume that the functions

under consideration do not share program labels, and that the function g has a unique parameter y.

The transformation is depicted in Figure 2.16, where Q stands for the assertion $\forall \mathsf{res}.\ \mathsf{post}(g)[{}^{x}/_{y^*}] \Rightarrow \mathsf{wp}_f(L_{ret})[{}^{\mathsf{res}}/_{r_d}]$. The graph G_g for function

$$
\begin{array}{ll}
\overline{f}[L_{call}] & \triangleq y := x,\ L_{call'} \\
\overline{f}[L_{call'}] & \triangleq (\mathsf{pre}(g) \wedge Q,\ \mathsf{nop},\ L_{sp_g}) \\
\overline{f}[L] & \triangleq [\![g[L]]\!] \quad \text{if } L \text{ is in } G_g \text{ domain} \\
\overline{f}[L_{ret'}] & \triangleq \mathsf{post}(g)[{}^{r_d}/_{\mathsf{res}}] \wedge Q, \mathsf{nop},\ L_{ret} \\
\text{where:} & \\
[\![(\phi,\ \mathsf{ins})]\!] & \triangleq (\phi \wedge Q,\ [\![\mathsf{ins}]\!]) \\
[\![\mathsf{return}\ r]\!] & \triangleq r_d := r,\ L_{ret'} \\
[\![\mathsf{ins}]\!] & \triangleq \mathsf{ins} \qquad \text{if ins} \neq \mathsf{return}
\end{array}
$$

Fig. 2.16. Function inlining transformation

g is inserted, with small changes, in replacement of the function call, and appropriate assignments are inserted before the function call. Notice that not only the return instructions are modified, but also the assertions, which are augmented with conditions (Q) that must be propagated through the complete set of instructions until the end, at label L_{ret}.

Certificate translation

The wp function is defined for the function call case propagating the conditions that must be satisfied when the function returns. The problem with function inlining is that this propagation is lost, when inserting an arbitrarily big piece of code that may contain assertions. For this reason and for the purpose of certificate generation, assertions are reinforced with the condition that we want to propagate (Q).

The assertion Q has the desired property that it cannot be modified by any instruction in the body of function g, assuming the disjointness of the set of local registers. In consequence, it is not difficult to prove in this case that the following property is satisfied:

$$\forall L \in \text{ domain of } G_g.\ \mathsf{wp}_g^{\mathtt{id}}(L) \wedge Q \Rightarrow \mathsf{wp}_{\overline{f}}^{\mathtt{id}}(L)$$

Notice that this property is similar to those we get when using an auxiliary function $f_{\mathcal{A}}$ to prove the results of the analysis, but in this case a certifying analyzer is not necessary since Q is always trivially preserved. Another property that is satisfied is

$$\forall L \in \text{ domain of } G_f.\ \mathsf{wp}_f(L) = \mathsf{wp}_{\overline{f}}(L)$$

From these two results and the definition of the transformation, we can reconstruct a new proof for the modified function $\overline{f}$. For every label L in the domain of G_f such that $f[L] = (\phi,\ \mathsf{ins})$, we have that $\overline{f}[L] = (\phi,\ \mathsf{ins})$ and also that $\mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L) = \mathsf{wp}^{\mathtt{id}}_{f}(L)$, so certificates are simply preserved in this range. For labels in the domain of G_g, if $g[L] = (\phi,\ \mathsf{ins})$ then $\overline{f}[L]$ is defined as $(\phi \wedge Q,\ \mathsf{ins})$, and using the proof for $\mathsf{wp}^{\mathtt{id}}_{g}(L) \wedge Q \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$, and the original proof for $\phi \Rightarrow \mathsf{wp}^{\mathtt{id}}_{g}(L)$ we get a certificate for $\phi \wedge Q \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$.

It remains to see the cases for the assertions introduced at labels $L_{ret'}$ and $L_{call'}$. In the latter, the certificate corresponding to the proof obligation related to the precondition of g is used. The former proof obligation is clearly provable as well.

Proof of auxiliary property.

Proof. We sketch here a proof for the properties stated above. The proof is performed by simultaneous induction with the order relation induced by the definition of *well-annotated* programs, for the following goals:

$$\forall L \in \text{ domain of } G_g.\ \mathsf{wp}_g(L) \wedge Q \Rightarrow \mathsf{wp}_{\overline{f}}(L)$$

and

$$\forall L \in \text{ domain of } G_g.\ \mathsf{wp}^{\mathtt{id}}_g(L) \wedge Q \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)\ .$$

— In case $g[L] = (\phi,\ \mathsf{ins})$ the first property is satisfied by definition and the second one can be proved by case analysis in ins.

— If $g[L] = r_1 := r_2,\ L'$ then, under the hypothesis that r_1 is not a free variable in Q, we get from the inductive hypothesis $\mathsf{wp}_g(L') \wedge Q \Rightarrow \mathsf{wp}_{\overline{f}}(L')$, a proof for $\mathsf{wp}^{\mathtt{id}}_g(L) \wedge Q \Rightarrow \mathsf{wp}^{\mathtt{id}}_{\overline{f}}(L)$ and $\mathsf{wp}_g(L) \wedge Q \Rightarrow \mathsf{wp}_{\overline{f}}(L)$.

— If $g[L] = b\ ?\ L_t : L_f$ then we have to prove $\mathsf{wp}_{\overline{f}}(L)$ with the hypothesis $(b \Rightarrow \mathsf{wp}_g(L_t) \wedge \neg b \Rightarrow \mathsf{wp}_g(L_f))$ and Q. By inductive hypothesis we have that $\mathsf{wp}_g(L_t) \wedge Q \Rightarrow \mathsf{wp}_{\overline{f}}(L_t)$ and that $\mathsf{wp}_g(L_f) \wedge Q \Rightarrow \mathsf{wp}_{\overline{f}}(L_f)$. Hence, we can construct a proof for $b \Rightarrow \mathsf{wp}_{\overline{f}}(L_t)$ and $\neg b \Rightarrow \mathsf{wp}_{\overline{f}}(L_f)$.

— In case $g[L] = \mathsf{return}\ r$ then $\mathsf{wp}_{\overline{f}}(L)$ is equal to $\mathsf{post}(g)[^{r}/_{\mathsf{res}}] \wedge Q$. And, since $\mathsf{wp}_g(L) = \mathsf{post}(g)[^{r}/_{\mathsf{res}}]$, the consequence is clearly satisfied.

— Since any other instruction is similar and does not represent interesting difficulties, a special treatment is not deserved.

2.3.10 Summary

In Section 2.2, we have showed that compiling programs from a high-level language to an RTL representation preserves proof obligations as long as no optimization is performed.

In this section, we have studied several standard compiler optimizations, and under each particular case, we have showed that proof obligations may be modified and, thus, that original certificates may not be reused. To solve this

difficulty, we have proposed a variety of techniques, some of them based on the existence of a certifying analyzer and some others solved by ad-hoc techniques. Optimizations that rely on a certifying analyzer range from constant propagation or copy propagation, to common subexpression elimination, or a loop optimization like induction variable strength reduction.

When dealing with dead register elimination we have showed a difficulty that arises when dead registers occur on invariants and we have proposed a transformation on both the program and the specifications called ghost variable introduction. Finally, we prove that, after applying this transformation, certificate translation for the modified proof obligations is quite simple.

We have shown that unreachable code elimination is trivial, and we have also proposed particular techniques to deal with function inlining and register allocation. The latter is in fact a simplified version of the standard phase of code generation, and relies on the simplifying assumption that dead variable elimination has already been performed.

3

An Abstract Interpretation Model of Certificate Translation

This article formalizes in the setting of abstract interpretation a method to transform certificates of program correctness along semantically justified program transformations.

3.1 Introduction

In this chapter, we take a more general approach and study certificate translation under the setting of abstract interpretation [26, 27]. The existence of certificate translators in Chapter 2 is studied with a tight integration to a particular program representation and verification environment. The lack of a framework in which to formulate the basic concepts of certificate translation was a clear limitation of our earlier work, and made it difficult to assess the generality of certificate translation. Therefore, the choice of an abstract framework provides a substantial leverage with respect to previous results. In particular, the abstract interpretation framework is a common model for the two key components of a certificate translation procedure: the verification environment in which the original certificate is produced and the static analysis that justifies a program optimization. A unifying theory to describe these components allows a more precise study of their interaction in a certificate translation process.

A common means to verify program properties is to consider (pre or post) fixpoints of the transfer functions of an abstract interpretation. On the other hand, to provide evidence to a code consumer that a program follows a specified policy, the code producer binds the program with a (partial) precomputed solution. The code client just needs to check that a labeling satisfies a set of inequalities to gain confidence about the program correctness. However, for some abstract domains, checking the constraints required for the solution of the analysis is too costly or even undecidable. That is the case, for instance, with the domain of polyhedra or the domain of logical formulae in which our

PCC is based. To solve this difficulty, a notion of certificate must be introduced to allow one to efficiently check that a constraint is satisfied. To model this scenario, we have provided a mild extension of the abstract interpretation model to incorporate a certificate infrastructure. We do not commit to a particular representation of certificates; instead, we have defined an abstract notion of proof algebra, that includes a set of functions, closed in the domain of certificates, that are used to define a certificate translator for each program transformation. One can see the VCgen framework defined in the previous chapter as particular instance of the extended abstract interpretation framework. As stated in Chapter 2, a certifying analyzer is a main component in the definition of a certificate translator. It is standard to state the soundness of an abstract interpretation with respect to the concrete semantics by means of a consistency relation between the concrete and abstract transfer functions. This notion of consistency extends not only to the concrete semantics but between abstract interpretations at different levels of abstractions. After extending an abstract interpretation framework with a certificate infrastructure, it is possible to certify the consistency between two abstract interpretations. We show in this chapter that proving the existence of (and defining) a certifying analyzer boils down to discharging the verification conditions required to prove consistency of the analysis with respect to the verification environment.

In this chapter we do not consider particular program optimizations, but arbitrary program transformations without assuming any advantage in terms of performance, code size, or resource consumption. We have studied certificate translation in the presence of basic transformations that, combined, can represent a wide range of program optimizations such as those presented in Chapter 2. This approach is relevant not only because it enables us to analyze arbitrary program optimizations, extending the range of certificate translation, but because it decomposes certificate translation into less complex tasks, and hence simpler proof obligations. To translate the certificate in the presence of these basic transformations, we require a relation between the transformed and the original programs. This relation is not expressed in the concrete semantics domain, as is the case in Certified Compilation, but as proof obligations on the abstract certificate infrastructure. This set of requirements can then be opportunely instantiated to a particular analysis and verification domain (and a particular transformation) to yield the proof obligations to be discharged in order to complete the definition of the certificate translator.

In Chapter 2, for an extensive class of program optimizations, we have proposed to strengthen annotations with the result of the analysis in order to reuse the original certificates. This is the case clearly for many common program optimizations, which rely on a safety analysis to justify semantics preserving local transformations. In the abstract lattice domain, strengthening annotations is defined via a corresponding *meet* operator. This technique cannot be applied, for instance with optimizations based on a liveness analysis, as is the case for dead variable elimination. In this situation, an ad-hoc

solution is proposed in Chapter 2 to define a certificate translator to deal with this transformation. The approach consists on the introduction of ghost variables and ghost assignments, a facility without computational role that assists specification and verification. Instead, in this chapter we provide a mild generalization of the technique based on invariant strengthening. It consists on merging the result of the analysis with the original specification by means of an abstract composition operator. The approach is general enough to model certificate translation for dead variable elimination, which entails (existentially) quantifying out dead variables from the intermediate assertions.

3.2 Program, Semantics, Abstract Interpretation

In this section, we formally introduce the abstract framework in which we study the existence of certificate translators. First, an abstract representation of programs is given with the associated semantics. Then, a general abstract interpretation framework is presented, to unify the main components of certificate translation: the analysis environment results that justifies program transformations and the verification infrastructure in which the original certificates are defined. Given a labeling from program points to abstract values, we provide a set of constraints whose solutions are a sound description of the reachable execution states, as long as the abstract interpretation is consistent with the semantics. A formal definition of this notion of consistency is also provided. To capture the existence of certificates in a Proof Carrying Code setting, an abstract algebra of certificates is given, and certificate infrastructures are defined as a mild extension of abstract interpretation frameworks.

3.2.1 Program, Semantics

The definition of programs is general enough to cover several syntactic program representations, including labeled structured languages, low level bytecode, and concurrent programming languages.

We represent programs as flow graphs. Thus, programs are directed pointed graphs with a distinguished set of output nodes, from which execution may not flow. The representation of programs as a graph includes also the representation of programs composed of several methods, allowing us to consider also interprocedural analyses in our framework.

We do not need to commit to a particular definition of execution states. Instead, we remain at an abstract level and let Env represent a set of environments. One instance of the domain Env for imperative languages is that of mappings from a domain of program variables (or memory locations) to primitive values. We define the domain of execution states as $\mathsf{State} = \mathcal{N} \times \mathsf{Env}$.

Definition 3.1 (Programs). *A program is a pointed directed graph P represented by the tuple $\langle \mathcal{N}, \mathcal{E}, \rightsquigarrow, l_{\mathsf{init}} \rangle$, where $\mathcal{N}$ is a set of nodes, $l_{\mathsf{init}} \in \mathcal{N}$ is a*

distinguished initial node, $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ a finitely branching relation (elements of $\mathcal{E}$ are called edges), and $\rightsquigarrow$ is a relation on $\mathcal{N} \times \mathsf{Env}$. We let $\mathcal{O} \subseteq \mathcal{N}$ be the set of nodes without successors, i.e., $l \in \mathcal{O}$ iff for all $l' \in \mathcal{N}$, $\langle l, l' \rangle \notin \mathcal{E}$. The semantics of program is defined from the relation $\rightsquigarrow \subseteq \mathsf{State} \times \mathsf{State}$ as a nondeterministic state transition

We say that the program P terminates execution in an environment η', starting from an environment η, if $\langle l_{\mathsf{init}}, \eta \rangle \rightsquigarrow^\star \langle l_o, \eta' \rangle$ for some $l_o \in \mathcal{O}$ ($\rightsquigarrow^\star$ represents the reflexive and transitive closure of the relation $\rightsquigarrow$).

Throughout this section, we let $P = \langle \mathcal{N}, \mathcal{E}, \rightsquigarrow, l_{\mathsf{init}} \rangle$ be a program.

Consider for instance the RTL language syntax of Figure 2.2. Given a sequence of instructions, we can associate a program node for each position on the code graph. Alternatively, a coarser grained representation may associate a node to the entry point of each basic block. For instance, the execution of the assignment $r_d := \mathsf{op},\ L'$ at a program node L reaches exactly one successor point, i.e., L', then a reliable program representation must contain the edge $\langle L, L' \rangle$. In the case of a branch instruction $\mathsf{cmp}\ ?\ L_t : L_f$ at node L, the edges $\langle L, L_t \rangle$ and $\langle L, L_f \rangle$ indicate that execution may follow the two distinct branches pointed by L_t and L_f.

Example 3.2.1 *Consider as a running example the fast exponentiation algorithm shown in Figure 3.1. Its representation as a (labeled) graph is given in Figure 3.2; edges represent either assignments of the form x:=e, in which case the node has exactly one successor, or branches of conditional statements of the form b?, in which case the node has exactly two successor nodes, corresponding to the true and false branch of the condition, respectively.*

```
l1 : c := 1
     x' := x
     y' := y
l2 : while (y' ≠ 1) do
l3 :   if (y' mod 2 = 1) then
l4 :     c := c × x'
       fi
l5 :   x', y' := x'^2, y'/2
     done
l6 : x' = x' × c
```

Fig. 3.1. Fast exponentiation algorithm.

3.2.2 Abstract Interpretations of Program Semantics

In contrast to standard abstract interpretation, we consider domains with a preorder relation, rather than partial orders. Recall that a binary relation

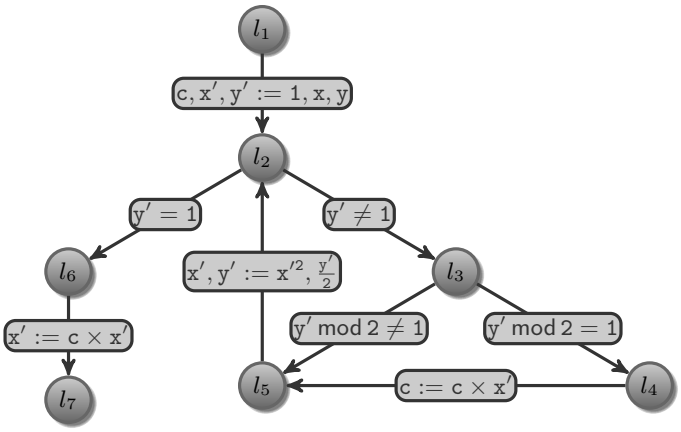


Fig. 3.2. Original Program Representation

$\sqsubseteq$ on a set A is a preorder if it is reflexive and transitive; a preorder is a partial order if it is also antisymmetric. The reason for this distinction, is that the natural domain for the verification infrastructure is that of propositions. We do not want to view it as a partial order since it would later imply (in Definition 3.8) that any formulas ϕ_1 and ϕ_2, if logically equivalent (i.e., if $\phi_1 \sqsubseteq \phi_2$ and $\phi_2 \sqsubseteq \phi_1$), they are identical by antisymmetry. Then, they may have the same certificates (since $\phi_1 = \phi_2$), which is not necessarily the case for any pair of equivalent formulas.

Definition 3.2 (Abstract interpretation). *Let $P = \langle \mathcal{N}, \mathcal{E}, \rightsquigarrow, l_{\mathsf{init}} \rangle$ be a program. An abstract interpretation of P is a triple $I = \langle \mathbf{A}, \{T_e\}_{e \in \mathcal{E}}, f \rangle$, where*

- **A** *is a lattice $\langle A, \sqsubseteq_A, \sqcup_A, \sqcap_A, \top_A, \bot_A \rangle$ of abstract states, with $\sqsubseteq_A$ a preorder in A. By abuse of notation we may omit the subscript $_A$;*
- *f is the flow sense, either forward ($f = \downarrow$), or backward ($f = \uparrow$);*
- *$\{T_e\}_{e \in \mathcal{E}} : A \to A$ is a family of monotone transfer functions.*

Thus, an abstraction of the program consists of an abstract domain, e.g., assertions or types, and a set of transfer functions, e.g., weakest precondition transformers. Although it is sufficient to consider meet or join semi-lattices, depending on the flow of the interpretation, we find it more convenient to require our domains to be lattices, since we deal both with forward and backwards analyses.

One particular abstract interpretation that describes the program semantics is $\langle \mathcal{P}(\mathsf{Env}), \mathsf{exec}_e, \downarrow \rangle$, where the forward transfer functions $\{\mathsf{exec}_e\}_{e \in \mathcal{E}}$ are defined for every $\langle l, l' \rangle \in \mathcal{E}$ and $X \subseteq \mathsf{Env}$ as $\mathsf{exec}_{\langle l,l' \rangle}(X) = \{\eta' \in \mathsf{Env} \mid \exists \eta \in X.\ \langle l, \eta \rangle \rightsquigarrow \langle l', \eta' \rangle\}$.Another instance of the abstract interpretation model is the verification framework defined in Chapter 2, where the abstract domain is

the lattice of logic formulae and the backward transfer functions are defined as weakest-precondition transformers.

Consider the lattice domains $\mathbf{A} = \langle A, \sqsubseteq, \sqcap, \sqcup, \top, \bot \rangle$ and $\mathbf{A}^\sharp = \langle A^\sharp, \sqsubseteq^\sharp, \sqcap^\sharp, \sqcup^\sharp, \top^\sharp, \bot^\sharp \rangle$. From the standard definition of Galois connection, we only require the existence of a monotone function $\gamma : A^\sharp \to A$. We refer to $\mathbf{A}$ and $\mathbf{A}^\sharp$ as the concrete and abstract domain, respectively, and to γ as the concretization function.

Definition 3.3 (Consistency). *Let $I^\sharp = \langle \mathbf{A}^\sharp, T_e^\sharp, f^\sharp \rangle$ and $I = \langle \mathbf{A}, T_e, f \rangle$ be abstract interpretations and $\gamma : A^\sharp \to A$ a concretization function. The abstract interpretation $I^\sharp = \langle \mathbf{A}^\sharp, T_e^\sharp, f^\sharp \rangle$ is consistent with $I = \langle \mathbf{A}, T_e, f \rangle$ iff for all $a \in A^\sharp$:*

- *if $f = f^\sharp = \downarrow$ then $T_e(\gamma(a)) \sqsubseteq \gamma(T_e^\sharp(a))$;*
- *if $f = f = \uparrow$ then $T_e(\gamma(a)) \sqsupseteq \gamma(T_e^\sharp(a))$;*
- *if $f = \downarrow$ and $f^\sharp = \uparrow$ then $T_e(\gamma(T_e^\sharp(a))) \sqsubseteq \gamma(a)$;*
- *if $f = \uparrow$ and $f^\sharp = \downarrow$ then $T_e(\gamma(T_e^\sharp(a))) \sqsupseteq \gamma(a)$;*

The requirements above express the fact that $I^\sharp$ is a *safe* approximation of the abstract interpretation I. Consider, for instance, when both $\{T_e\}_{e \in \mathcal{E}}$ and $\{T_e^\sharp\}_{e \in \mathcal{E}}$ are sets of forward transfer functions, i.e, $f = f^\sharp = \downarrow$. Then, the well-known consistency requirement $T_e(\gamma(a)) \sqsubseteq \gamma(T_e^\sharp(a))$ states that the execution step $T_e^\sharp$ from the abstract value a returns a value that is less precise than doing the same with the more concrete transition represented by T_e. Another common example is the consistency required between a weakest precondition-based analysis and the concrete semantics, expressing precisely the soundness of the verification environment.

In the rest of the chapter, we view both the analysis performed by the compiler and the verification framework as abstract interpretations that are consistent with $\langle \mathcal{P}(\mathsf{Env}), \mathsf{exec}_e, \downarrow \rangle$. Furthermore, in the following sections we require the analysis performed by a compiler to be a consistent abstraction of the verification framework.

Given an abstract lattice $\mathbf{A}$, and a concretization function $\gamma : A \to \mathcal{P}(\mathsf{Env})$, we define a satisfaction relation $\models_A \subseteq \mathsf{Env} \times A$ as

$$\models_A \eta : a \stackrel{\text{def}}{=} \eta \in \gamma(a) \ .$$

Intuitively, we interpret $\models_A \eta : a$ as the fact that the environment η satisfies the property a. The relation $\sqsubseteq_A$ is an approximation order, since by the monotonicity of γ, if $\models_A \eta : a$ and $a \sqsubseteq_A a'$ then $\models_A \eta : a'$. In the following, we simply write $\models$ omitting the subscript A.

In the context of proof carrying code, one such underlying lattice is that of logical assertions with logical implication $\Rightarrow$ as preorder, and the transfer functions are the predicate transformers (based on weakest precondition or strongest postcondition) induced by instructions at any given program point.

Consider the $\mathsf{wp}_f^{\mathtt{id}}$ function used to define the VCgen for RTL code in Chapter 2 (Figure 2.4). One can see the $\mathsf{wp}_f^{\mathtt{id}}$ function as an instance of a set

of transfer functions $\{\mathsf{wp}_e\}_{e\in\mathcal{E}}$ over the domain of logical formulae, and given in terms of the instructions located at the program points designated by the edge e. As is standard, the backward transfer function $\mathsf{wp}_{\langle l,l+1\rangle}(\phi)$ is defined as the substitution of the expression e for x ($\phi[{}^{e}/_{x}]$) when the instruction at the edge $\langle l, l+1\rangle$ is an assignment of the form $x := e$ (i.e., $\mathsf{wp}^{\mathtt{id}}_f(x := e, L) = \mathsf{wp}_f(L)[{}^{e}/_{x}]$). When considering a conditional jump instruction $\mathsf{cmp}\ ?\ l_t : l_f$, we define $\mathsf{wp}_{\langle l,l_t\rangle}(\phi)$ and $\mathsf{wp}_{\langle l,l_f\rangle}(\psi)$ as $\mathsf{cmp} \Rightarrow \phi$ and $\neg\mathsf{cmp} \Rightarrow \psi$, respectively. In Figure 2.4 the function $\mathsf{wp}^{\mathtt{id}}_f(\mathsf{cmp}\ ?\ l_t : l_f)$ merges both results with the corresponding meet operator, i.e., the logical $\wedge$. From Definition 3.3, to show that the verification environment is consistent with the program semantics we must prove for every $e \in \mathcal{E}$ and any assertion ϕ, that $\mathsf{exec}_e(\gamma(\mathsf{wp}_e(\phi))) \subseteq \gamma(\phi)$. In the case of the assignment $x := e$, this corresponds to showing that for any η in Env s.t. $\eta \models \phi[{}^{e}/_{x}]$ we have $[\eta \mid x \mapsto n] \models \phi$, where n is the value of the expression e in η and for any function f, $[f \mid x \mapsto n]$ denotes the function that maps x to n and y to fy if $y \neq x$.

Solutions

A common means to verify program properties is to consider (pre or post) fixpoints of the transfer functions of an abstract interpretation.

Definition 3.4 (Solution). *A labeling $S : \mathcal{N} \to A$ is a solution of the abstract interpretation I if*

- *$f =\uparrow$ and for every l in $\mathcal{N}$, $S(l) \sqsubseteq \bigsqcap_{\langle l,l'\rangle\in\mathcal{E}} T_{\langle l,l'\rangle}(S(l'))$;*
- *$f =\downarrow$ and for every node l in $\mathcal{N}$, $S(l) \sqsupseteq \bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T_{\langle l',l\rangle}(S(l'))$.*

Notice that $\bigsqcap_{e\in\mathcal{E}}$ represents a finite number of applications of the operation $\sqcap$. Alternatively, we may require for every edge $\langle l, l'\rangle$ that $S(l) \sqsubseteq T_{\langle l,l'\rangle}(S(l'))$ or $T_{\langle l,l'\rangle}(S(l)) \sqsubseteq S(l')$ for $f =\uparrow$ or $f =\downarrow$, respectively.

Since we are interested in describing the set of states that may reach each program point, we consider a labeling $S : \mathcal{N} \to \mathcal{P}(\mathsf{Env})$ that associates a set of environments with every program node. The labeling S is a solution of $\langle \mathcal{P}(\mathsf{Env}), \mathsf{exec}_e, \downarrow\rangle$ if for every $\langle l, l'\rangle \in \mathcal{E}$, $\mathsf{exec}_{\langle l,l'\rangle}(S(l)) \subseteq S(l')$. The following result states that if S is a solution of $\langle \mathcal{P}(\mathsf{Env}), \{\mathsf{exec}_e\}_{e\in\mathcal{E}}, \downarrow\rangle$, every execution starting with an environment in $S(l_{\mathsf{init}})$ only reaches states $\langle l, \eta\rangle$ with $\eta \in S(l)$.

Proposition 3.5. *Let $P = \langle \mathcal{N}, \mathcal{E}, \rightsquigarrow, l_{\mathsf{init}}\rangle$ and $S : \mathcal{N} \to \mathcal{P}(\mathsf{Env})$ a solution of $\langle \mathcal{P}(\mathsf{Env}), \mathsf{exec}_e, \downarrow\rangle$. For every $l, l' \in \mathcal{N}$, $\eta, \eta' \in \mathsf{Env}$ such that $\langle l, \eta\rangle \rightsquigarrow^\star \langle l', \eta'\rangle$, if $\eta \in S(l)$ then $\eta' \in S(l')$ (where $\rightsquigarrow^\star$ denotes the reflexive and transitive closure of $\rightsquigarrow$).*

Proof. The proof assumes as hypothesis that $\langle l, \eta\rangle \rightsquigarrow^k \langle l', \eta'\rangle$ and proceeds by natural induction on k.

The following two results state the soundness of an abstract interpretation with respect to the concrete semantics of a program.

Proposition 3.6. *Let $I = \langle A, \{T_e\}_{e\in\mathcal{E}}, f\rangle$ and $I^\sharp = \langle A^\sharp, \{T_e^\sharp\}_{e\in\mathcal{E}}, f^\sharp\rangle$ be abstract interpretations of program P, such that $I^\sharp$ is consistent with I. If the labeling $S^\sharp : \mathcal{N} \to A^\sharp$ is a solution of $I^\sharp$, then the labeling $S : \mathcal{N} \to A$ defined as $S(l) = \gamma(S^\sharp(l))$ is a solution of I.*

Proof. Consider for instance the case $f = \uparrow$ and $f^\sharp = \downarrow$ (the other cases are symmetric). Since $S^\sharp$ is a solution of $I^\sharp$ we have for any $l' \in \mathcal{N}$:

$$S^\sharp(l') \sqsupseteq \bigsqcup_{\langle l,l'\rangle\in\mathcal{E}} T^\sharp_{\langle l,l'\rangle}(S^\sharp(l))$$

or, equivalently, for every edge $\langle l, l'\rangle \in \mathcal{E}$:

$$S^\sharp(l') \sqsupseteq T^\sharp_{\langle l,l'\rangle}(S^\sharp(l)) \ .$$

From the monotony of γ and $T_{\langle l,l'\rangle}$ we have then

$$T_{\langle l,l'\rangle}(\gamma(S^\sharp(l'))) \sqsupseteq T_{\langle l,l'\rangle}(\gamma(T^\sharp_{\langle l,l'\rangle}(S^\sharp(l))))$$

which together with the consistency of $I^\sharp$ w.r.t. I (i.e., $\gamma(a) \sqsubseteq T_e(\gamma(T_e^\sharp(a)))$ for any $a \in A$ and $e \in \mathcal{E}$) implies the condition we want to prove:

$$T_{\langle l,l'\rangle}(\gamma(S^\sharp(l'))) \sqsupseteq \gamma(S^\sharp(l)) \ .$$

Lemma 3.7. *Let S be a solution of the abstract interpretation $I = \langle A, \{T_e\}, f\rangle$ and assume I consistent with $\langle \mathcal{P}(\mathsf{Env}), \mathsf{exec}_e, \downarrow\rangle$. Then, if $\langle l, \eta\rangle \rightsquigarrow^\star \langle l', \eta'\rangle$ and $\models \eta : S(l)$ then $\models \eta' : S(l')$.*

Proof. This lemma follows by application of Propositions 3.5 and 3.6.

3.2.3 Certified Solutions

A common means to provide evidence to a code consumer that a program follows a specified policy is to bind the program with a (partial) precomputed solution. The code client just needs to check that a labeling satisfies a set of inequalities to gain confidence about the program correctness. Abstraction Carrying Code (ACC) is an instance of PCC where programs come with a solution in an abstract interpretation that can be used to specify the consumer policy [1].

However, for some abstract domains, checking that the relation $\sqsubseteq$ holds may be undecidable or too costly. One may view our notion of certified solution as a natural extension of ACC to settings where the preorder relation is either undecidable, or expensive to compute, and where the use of certificates is required in order to check solutions. That is the case for instance of the lattice of logical formulae or for polyhedron analysis. Besson *et al.* [19] have recently developed a program analysis framework in which certificates are used to

verify inclusions between elements of the abstract domain of polyhedra. Their analysis is also an instance of a certified solution.

This section extends the basic framework of abstract interpretation with certificate infrastructures, in order to introduce formally the notion of certified solution.

Definition 3.9 provides a general definition of certified solution that is of independent interest from certificate transformation, and provides a unifying framework for existing *ad hoc* definitions, such as a weakest precondition based VCgens, and certified Abstraction Carrying Code [18]. For our purposes, one can think about certified solutions as:

- programs annotated with logical assertions, and bundled with a certificate of the correctness of the verification conditions, or
- programs annotated with abstract values (or types), and bundled with a certificate that the program is correct with respect to the interpretation of the abstract values.

In order to capture the notion of certified solution at an appropriate level of abstraction, we rely on a general notion of certificate infrastructure.

Definition 3.8 (Certificate infrastructure). *A certificate infrastructure for P consists of an abstract interpretation $I = \langle A, \{T_e\}_{e\in\mathcal{E}}, f\rangle$ for P, and a proof algebra $\mathcal{C}$ that assigns to every $a, a' \in A$ a set of certificates $\mathcal{C}(\vdash a \sqsubseteq a')$ s.t.:*

- *$\mathcal{C}$ is closed under the operations of Figure 3.3, where $a, b, c \in A$;*
- *$\mathcal{C}$ is sound, i.e., for every $a, a' \in A$, if $a \not\sqsubseteq a'$, then $\mathcal{C}(\vdash a \sqsubseteq a') = \emptyset$.*

In the sequel, we write $c :\vdash a \sqsubseteq a'$ or $c :\vdash a' \sqsupseteq a$ instead of $c \in \mathcal{C}(\vdash a \sqsubseteq a')$. For notational convenience, in the proofs we use the function

$$\mathsf{trans} : \mathcal{C}(\vdash a \sqsubseteq b) \to \mathcal{C}(\vdash b \sqsubseteq c) \to \mathcal{C}(\vdash a \sqsubseteq c)$$

defined by composition of the algebra operations $\mathsf{weak}_\sqcap$ *and* $\mathsf{elim}_\sqcap$.

In the context of proof carrying code, with the underlying lattice of logical assertions, it may be helpful for the reader to think about certificates in terms of the Curry-Howard isomorphism and consider that $\mathcal{C}$ is given by the typing judgment in a dependently typed λ-calculus, i.e., $\mathcal{C}(\phi) = \{e \in \mathcal{E} \mid \langle\rangle \vdash e : \phi\}$, where $\mathcal{E}$ is the set of expressions of the type theory. Under such assumptions, one can provide an obvious type-theoretical interpretation to the functions of Figure 3.3; for example, $\mathsf{intro}_\sqcap$ is given by the λ-term $\lambda f.\ \lambda g.\ \lambda a.\ \langle fa, ga\rangle$.

In the sequel, we let $I = \langle A, \{T_e\}, f\rangle$ be a certificate infrastructure for P.

Definition 3.9 (Certified solution). *A certified solution for I is a pair $\langle S, \vec{c}\rangle$, where $S : \mathcal{N} \to A$ is a labeling and $\vec{c} = (c_l)_{l\in\mathcal{N}}$ is a family of certificates s.t. for every $l \in \mathcal{N}$,*

- *if $f = \uparrow$ then $c_l :\vdash S(l) \sqsubseteq \bigsqcap_{\langle l,l'\rangle\in\mathcal{E}} T_{\langle l,l'\rangle}(S(l'))$;*

- *if* $f = \downarrow$ *then* $c_l :\vdash \bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T_{\langle l',l\rangle}(S(l')) \sqsubseteq S(l)$.

It follows from the soundness of the proof algebra that S is a solution for I. Many techniques, including lightweight bytecode verification and abstraction

$$
\begin{array}{l}
\mathsf{axiom} \;: \mathcal{C}(\vdash a \sqsubseteq a) \\
\mathsf{weak}_{\sqcap} : \mathcal{C}(\vdash a \sqsubseteq b) \to \mathcal{C}(\vdash a \sqcap c \sqsubseteq b) \\
\mathsf{weak}_{\sqcup} : \mathcal{C}(\vdash a \sqsubseteq b) \to \mathcal{C}(\vdash a \sqsubseteq b \sqcup c) \\
\mathsf{elim}_{\sqcap} \;: \mathcal{C}(\vdash c \sqcap a \sqsubseteq b) \to \mathcal{C}(\vdash c \sqsubseteq a) \to \mathcal{C}(\vdash c \sqsubseteq b) \\
\mathsf{intro}_{\sqcup} : \mathcal{C}(\vdash a \sqsubseteq c) \to \mathcal{C}(\vdash b \sqsubseteq c) \to \mathcal{C}(\vdash a \sqcup b \sqsubseteq c) \\
\mathsf{intro}_{\sqcap} : \mathcal{C}(\vdash a \sqsubseteq b) \to \mathcal{C}(\vdash a \sqsubseteq c) \to \mathcal{C}(\vdash a \sqsubseteq b \sqcap c)
\end{array}
$$

Fig. 3.3. Proof Algebra

carrying code, do not bundle code with a full (certified) solution, but with a partial labeling (and some certificates) from which a full (certified) solution can be reconstructed. The remaining of this section explains the construction of a (certified) solution from a (certified) partial labeling.

Definition 3.10 (Labeling). *A partial labeling is a partial function* $S : \mathcal{N} \rightharpoonup A$ *s.t. entry and output nodes are annotated, i.e.,* $\mathcal{O} \cup \{l_{\mathsf{init}}\} \subseteq \mathsf{dom}(S)$*, and such that the program is sufficiently annotated, i.e., the restriction* $P_{\mathcal{N}\setminus\mathsf{dom}(S)}$ *of* P *to nodes that are not annotated is acyclic. A labeling* S *is total if* $\mathsf{dom}(S) = \mathcal{N}$.

In a partial labeling annot, annotations on entry and output nodes serve as specification, whereas we need sufficient annotations on loops to reconstruct a total labeling $\overline{\mathsf{annot}}$ from the partial one.

Definition 3.11 (Annotation propagation, verification condition). *Let* annot *be a partial labeling. The labeling* $\overline{\mathsf{annot}}$ *is defined by the clause:*

- *if* $f = \uparrow$, $\overline{\mathsf{annot}}(l) = \begin{cases} \mathsf{annot}(l) & \text{if } l \in \mathsf{dom}(\mathsf{annot}) \\ \bigsqcap_{\langle l,l'\rangle\in\mathcal{E}} T_{\langle l,l'\rangle}(\overline{\mathsf{annot}}(l')) & \text{otherwise} \end{cases}$
- *if* $f = \downarrow$, $\overline{\mathsf{annot}}(l) = \begin{cases} \mathsf{annot}(l) & \text{if } l \in \mathsf{dom}(\mathsf{annot}) \\ \bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T_{\langle l',l\rangle}(\overline{\mathsf{annot}}(l')) & \text{otherwise} \end{cases}$

For every $l \in \mathsf{dom}(\mathsf{annot})$*, the verification condition* $\mathsf{vc}(l)$ *is defined by the clause*

- $\mathsf{vc}(l) := \mathsf{annot}(l) \sqsubseteq \bigsqcap_{\langle l,l'\rangle\in\mathcal{E}} T_{\langle l,l'\rangle}(\overline{\mathsf{annot}}(l'))$ *if* $f = \uparrow$;
- $\mathsf{vc}(l) := \bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T_{\langle l',l\rangle}(\overline{\mathsf{annot}}(l')) \sqsubseteq \mathsf{annot}(l)$ *if* $f = \downarrow$.

Given a partial labeling annot, one can build a certificate for $\overline{\mathsf{annot}}$ from certificates for the verification conditions on $\mathsf{dom}(\mathsf{annot})$.

Lemma 3.12. *Let* annot *be a partial labeling for* I *and assume we have a certificate* $c_l :\vdash \mathsf{vc}(l)$ *for every* $l \in \mathsf{dom}(\mathsf{annot})$*. Then there exists* $\vec{c'}$ *s.t.* $\langle \overline{\mathsf{annot}}, \vec{c'} \rangle$ *is a certified solution.*

Proof. By definition of $\overline{\mathsf{annot}}$*, one sees that* $\vec{c'}$ *defined as*

$$\vec{c'}(l) = \begin{cases} \vec{c}(l) & \text{if } l \in \mathsf{dom}(\vec{c}) \\ \mathsf{axiom}(\overline{\mathsf{annot}}(l)) & \text{otherwise} \end{cases}$$

is such that $\langle \overline{\mathsf{annot}}, \vec{c'} \rangle$ *is a certified solution.*

In the sequel, we shall abuse language and speak about certified solutions of the form $\langle \mathsf{annot}, \vec{c} \rangle$ where annot is a partial labeling and $\vec{c}$ is an indexed family of certificates that establish all verification conditions of annot.

Corollary 3.13. *Let* $\langle \mathsf{annot}, \vec{c} \rangle$ *be a certified partial labeling of* $\langle I, \mathcal{C} \rangle$ *and assume* I *consistent with the semantics of* P*. Then, if* $\langle l_{\mathsf{init}}, \eta \rangle \rightsquigarrow^* \langle l_o, \eta' \rangle$ *with* $l_o \in \mathcal{O}$ *and* $\models \eta : \mathsf{annot}(l_{\mathsf{init}})$ *then* $\models \eta' : \mathsf{annot}(l_o)$*.*

Proof. This result follows from Lemma 3.7, Lemma 3.12, and soundness of the certificate infrastructure.

Example 3.2.2 *The verification infrastructure to certify the running example is built from a weakest precondition calculus over first-order formulae. That is, the backward transfer functions are defined, for any assertion* ϕ*, as* $T_{\langle l,l' \rangle}(\phi) = \phi[^e/_x]$ *in case the node* l *contains the assignment* $x{:=}e$*, and as* $b \Rightarrow \phi$ *or* $\neg b \Rightarrow \phi$ *respectively for the positive and negative branch of a jump statement conditioned by the Boolean expression* b*. We as specification a partial labeling s.t.* $\mathsf{annot}(l_1) = \mathsf{true}$*, the invariant* $\mathsf{annot}(l_2)$ *is* $c \times x'^{y'} = x^y$ *and the postcondition* $\mathsf{annot}(l_7)$ *is* $x'{=}x^y$ *(as shown in Figure 3.7). We assume also that the functional specification* annot *is certified, i.e., the existence of a certified solution* $\langle \mathsf{annot}, \vec{c} \rangle$ *of* $I = \langle A, \{T_e\}, \uparrow \rangle$*.*

3.3 Certifying Analyzers

Commonly, program optimizations are performed in two steps. First an analysis detects static information from the program representation. Then, in the basis of this information, a semantic preserving transformation optimizes the code. Let the labeling $S : \mathcal{N} \rightharpoonup D$ represent verification annotations, for instance, a mapping from program nodes to the domain of logical formulae, and let the labeling $S^\sharp : \mathcal{N} \rightharpoonup D^\sharp$ represent the result of the analysis that justifies a transformation. Consider the programs P and P', where P' is the result of optimizing P based on the information provided by the labeling $S^\sharp$. Assume that S is a solution for the program P. Even if the validity of $S^\sharp$ indicates that the transformation does not alter the program semantics, we cannot conclude that the labeling S is a solution for the program P'. Indeed, several examples

in the previous chapter show that invariants are not preserved alongside simple optimizations such as constant propagation and common subexpression elimination.

As explained in the next section, to translate certificates, we must integrate the original labeling S with the interpretation of $S^\sharp$ in the domain D. To that end, we assume the existence of a concretization function $\gamma : D^\sharp \to D$. In addition, one must incorporate a certificate of the labeling $\gamma \circ S^\sharp$ with the original certificate of the labeling S.

In this section, we provide sufficient conditions to produce a certified labeling $\langle \gamma \circ S^\sharp, \mathsf{cert}' \rangle$ in the domain of the verification environment from any labeling $S^\sharp$ in the domain of the analysis. In the next section, we explain how to integrate this certified analysis result to achieve a transformation of the original certified solution $\langle S, \mathsf{cert} \rangle$ in the presence of program transformations.

Proposition 3.14 below generalizes a previous result of Chaieb [24], who only considered the case where $f = \uparrow$ and $f^\sharp = \downarrow$.

In the rest of this chapter, let P be a program, $I^\sharp = \langle \mathbf{A}^\sharp, \{T_e^\sharp\}, f^\sharp \rangle$ be an abstract interpretation for P, $I = \langle \mathbf{A}, \{T_e\}, f \rangle$ a certificate infrastructure of program P, and $\gamma : A^\sharp \to A$ a concretization function.

The following result is essential for a significant set of certificate translators. Suppose $I^\sharp$ represents a analysis framework over a decidable domain $\mathbf{A}^\sharp$ and I a verification environment with a notion of certificates. The proposition below states that not only a result S of the analysis I can be represented as a program annotation $\gamma \circ S$ in the domain of the verification framework, but that, under certain requirements, there is a procedure to generate a certificate for the labelling $\gamma \circ S$.

Proposition 3.14 (Existence of certifying analyzers). *For every solution S of $I^\sharp$, one can compute $\vec{c}$ s.t. $\langle \gamma \circ S, \vec{c} \rangle$ is a certified solution for I, provided there exist:*

- *for every $a, a' \in A^\sharp$ s.t. $a \sqsubseteq^\sharp a'$, the certificates $\mathsf{monot}_\gamma(a, a') :\vdash \gamma(a) \sqsubseteq \gamma(a')$ and $\mathsf{monot}_T(a, a') :\vdash T(a) \sqsubseteq T(a')$;*
- *for every $x \in A^\sharp$, a certificate $\mathsf{cons}(x)$, where $\mathsf{cons}(x)$ is defined in Figure 3.4 according to the flows of the abstract interpretations.*

Proof.

The proof proceeds by showing a step-by-step construction of the certificates for $\gamma \circ S$ from the certificates assumed by hypothesis and applying the functions of the proof algebra.

$$f = f^\sharp = \downarrow \quad \mathsf{cons}(x) :\vdash T_e(\gamma(x)) \sqsubseteq \gamma(T_e^\sharp(x))$$

$$f = f^\sharp = \uparrow \quad \mathsf{cons}(x) :\vdash T_e(\gamma(x)) \sqsupseteq \gamma(T_e^\sharp(x))$$

$$f = \uparrow,\ f^\sharp = \downarrow \ \mathsf{cons}(x) :\vdash T_e(\gamma(T_e^\sharp(x))) \sqsupseteq \gamma(x)$$

$$f = \downarrow,\ f^\sharp = \uparrow \ \mathsf{cons}(x) :\vdash T_e(\gamma(T_e^\sharp(x))) \sqsubseteq \gamma(x)$$

Fig. 3.4. Definition of $\mathsf{cons}(x)$

$f = f^\sharp = \downarrow$

$\mathsf{hyp} := T^\sharp_{\langle l',l\rangle}(S(l')) \sqsubseteq S(l)$
$p_1 := \mathsf{monot}_\gamma(\mathsf{hyp}) :\vdash \gamma(T^\sharp_{\langle l',l\rangle}(S(l'))) \sqsubseteq \gamma(S(l))$
$p_2 := \mathsf{cons}(S(l')) :\vdash T_{\langle l',l\rangle}(\gamma(S(l'))) \sqsubseteq \gamma(T^\sharp_{\langle l',l\rangle}(S(l')))$
$p_3 := \mathsf{weak}_\sqcap(-, p_1) :\vdash \gamma(T^\sharp_{\langle l',l\rangle}(S(l'))) \sqcap T_{\langle l',l\rangle}(\gamma(S(l'))) \sqsubseteq \gamma(S(l))$
$p_4 := \mathsf{elim}_\sqcap(p_3, p_2) :\vdash T_{\langle l',l\rangle}(\gamma(S(l'))) \sqsubseteq \gamma(S(l))$
$\vec{c}_l := \mathsf{intro}_\sqcup(\{p_4\}_{\langle l',l\rangle\in\mathcal{E}}) :\vdash \bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T_{\langle l',l\rangle}(\gamma(S(l'))) \sqsubseteq \gamma(S(l))$

$f = f^\sharp = \uparrow$

$\mathsf{hyp} := S(l) \sqsubseteq^\sharp T^\sharp_{\langle l,l'\rangle}(S(l'))$
$p_1 := \mathsf{monot}_\gamma(\mathsf{hyp}) :\vdash \gamma(S(l)) \sqsubseteq \gamma(T^\sharp_{\langle l,l'\rangle}(S(l')))$
$p_2 := \mathsf{cons}(S(l')) :\vdash \gamma(T^\sharp_{\langle l,l'\rangle}(S(l'))) \sqsubseteq T_{\langle l,l'\rangle}(\gamma(S(l')))$
$p_3 := \mathsf{trans}(p_1, p_2) :\vdash \gamma(S(l)) \sqsubseteq T_{\langle l,l'\rangle}(\gamma(S(l')))$
$\vec{c}_l := \mathsf{intro}_\sqcap(\{p_4\}_{\langle l,l'\rangle\in\mathcal{E}}) :\vdash \gamma(S(l)) \sqsubseteq \bigsqcap_{\langle l,l'\rangle\in\mathcal{E}} T_{\langle l,l'\rangle}(\gamma(S(l')))$

$f = \uparrow$ *and* $f^\sharp = \downarrow$

$\mathsf{hyp} := T^\sharp_{\langle l',l\rangle}(S(l')) \sqsubseteq^\sharp S(l)$
$p_1 := \mathsf{monot}_\gamma(\mathsf{hyp}) :\vdash \gamma(T^\sharp_{\langle l',l\rangle}(S(l'))) \sqsubseteq \gamma(S(l))$
$p_2 := \mathsf{monot}_T :\vdash T_{\langle l',l\rangle}(\gamma(T^\sharp_{\langle l',l\rangle}(S(l')))) \sqsubseteq T_{\langle l',l\rangle}((\gamma(S(l)))$
$p_3 := \mathsf{cons}(S(l')) :\vdash \gamma(S(l')) \sqsubseteq T_{\langle l',l\rangle}(\gamma(T^\sharp_{\langle l',l\rangle}(S(l'))))$
$p_4 := \mathsf{trans}(p_3, p_2) :\vdash \gamma(S(l')) \sqsubseteq T_{\langle l',l\rangle}(\gamma(S(l)))$
$\vec{c}_{l'} := \mathsf{intro}_\sqcap(\{p_4\}_{\langle l',l\rangle\in\mathcal{E}}) :\vdash \gamma(S(l')) \sqsubseteq \bigsqcap_{\langle l',l\rangle\in\mathcal{E}} T_{\langle l',l\rangle}(\gamma(S(l)))$

$f = \downarrow$ *and* $f^\sharp = \uparrow$

$\mathsf{hyp} := S(l) \sqsubseteq^\sharp T^\sharp_{\langle l,l'\rangle}(S(l'))$
$p_1 := \mathsf{monot}_\gamma(\mathsf{hyp}) :\vdash \gamma(S(l)) \sqsubseteq \gamma(T^\sharp_{\langle l,l'\rangle}(S(l')))$
$p_2 := \mathsf{monot}_T :\vdash T_{\langle l,l'\rangle}(\gamma(S(l))) \sqsubseteq T_{\langle l,l'\rangle}(\gamma(T^\sharp_{\langle l,l'\rangle}(S(l'))))$
$p_3 := \mathsf{cons}(S(l')) :\vdash T_{\langle l,l'\rangle}(\gamma(T^\sharp_{\langle l,l'\rangle}(S(l')))) \sqsubseteq \gamma(S(l'))$
$p_4 := \mathsf{trans}(p_3, p_2) :\vdash T_{\langle l',l\rangle}(\gamma(S(l))) \sqsubseteq \gamma(S(l'))$
$\vec{c}_{l'} := \mathsf{intro}_\sqcup(\{p_4\}_{\langle l,l'\rangle\in\mathcal{E}}) :\vdash \bigsqcup_{\langle l,l'\rangle\in\mathcal{E}} T_{\langle l,l'\rangle}(\gamma(S(l))) \sqsubseteq \gamma(S(l'))$

While Proposition 3.14 provides a means to construct certifying analyzers, it is sometimes of interest to rely on more direct methods to generate certificates: in [7], we show how to construct compact certificates for constant propagation and common sub-expression elimination in an intermediate language.

3.4 Certificate Translation

In this section, we provide sufficient conditions for the existence for certificate translators that map certificates of a program P into certificates of another program P', derived from P by a program transformation. Rather than attempting to prove a general result where P and P' are related in some complex manner, we establish three results for basic transformations that can be used in combination to cover many cases of interest.

In a first instance, Section 3.4.1 considers a program transformation that consists in duplicating fragments of the graph representation of P, as is the case for transformations such as loop peeling and function inlining. In a second instance, certificate transformation as defined in Section 3.4.2 requires that the transformed program P' is a subgraph of the original program P. This is the case, for example, when P' is derived from P by applying optimizations such as constant propagation or common sub-expression elimination. In a third instance, in Section 3.4.3, we abstract away some of the structure of the program to deal, in combination with other basic transformations, with optimizations that do not preserve so tightly the structure of programs, such as code motion. Finally, in Section 3.4.4, we generalize certificate translation, covering optimizations such as dead variable elimination.

3.4.1 Code Duplication

In this section, we consider the case where some subgraphs of the initial program are duplicated in the transformed program, mainly with the aim to enable further program optimizations. Typical cases of code duplication are loop peeling (unrolling) and function inlining. Consider for instance the case of loop peeling, in which one or more iterations of the loop body are executed separately before entering the loop. In the piece of code on the left, we duplicate one iteration of the statement c representing the loop body. It is executed under the condition b in order to preserve the original program semantics.

```
x := 0;
while b do c
```

```
x := 0;
if b then
  c;
  while b do c
fi
```

After the transformation, the first occurrence of c can benefit from the fact that x is equal to 0, a condition that may be invalidated by the successive loop

iterations. To represent this program transformation in our abstract setting, consider the Figure 3.5. The graph on the left shows the original program, where node l_2 represents the head of a loop, and the edge $\langle l_2, l_2\rangle$ the execution of the loop body. The graph on right left shows the result of peeling the first execution of the loop body. When execution reaches node l'_2, the transition $\langle l'_2, l_2\rangle$ (representing one execution of the loop body) is taken if the loop guard is satisfied. Otherwise execution continues to node l_3.

Fig. 3.5. Loop Peeling Example

In the rest of this section, we formalize code duplication in terms of our abstract program representation, and we show how to transform certificates after applying this program representation transformation.

Definition 3.15 (Node replication). *A program* $P^+ = \langle \mathcal{N} \cup \mathcal{N}^+, \mathcal{E}^+, \rightsquigarrow^+, l_{\mathsf{init}}\rangle$ *is the result of replicating nodes of program* $P = \langle \mathcal{N}, \mathcal{E}, \rightsquigarrow, l_{\mathsf{init}}\rangle$ *if*

- $\mathcal{N}^+ \subseteq \{l^+ \mid l \in \mathcal{N}\}$*;*
- *for every* $l_1, l_2 \in \mathcal{N}$*, if* $\langle l_1^+, l_2\rangle$*,* $\langle l_1, l_2^+\rangle$*, or* $\langle l_1^+, l_2^+\rangle$ *is in* $\mathcal{E}^+$ *then* $\langle l_1, l_2\rangle$ *is in* $\mathcal{E}$*, i.e., subgraph duplication preserves the structure; and*
- *the semantics relation* $\rightsquigarrow^+$ *is such that for every* $\langle l, l'\rangle \in (\mathcal{E}^+ \cap (\{l_1, l_1^+\} \times \{l_2, l_2^+\}))$*,* $\langle l, \eta\rangle \rightsquigarrow^+ \langle l', \eta'\rangle$ *iff* $\langle l_1, \eta\rangle \rightsquigarrow \langle l_2, \eta'\rangle$*.*

From the definition, a sequence $\langle \bar{l_1}, \bar{l_2}\rangle, \langle \bar{l_2}, \bar{l_3}\rangle, \ldots, \langle \bar{l_{k-1}}, \bar{l_k}\rangle$ with $l_i \in \{l_i, l_i^+\}$ is a path in P^+ only if $\langle l_1, l_2\rangle, \langle l_2, l_3\rangle, \ldots, \langle l_{k-1}, l_k\rangle$ is a sequence in P.

Let $\langle I, \mathcal{C}\rangle$ be a certificate infrastructure with $I = \langle \mathbf{A}, \{T_e\}_{e\in\mathcal{E}}, f\rangle$. Then, we define an extended certificate infrastructure $I^+ = \langle \mathbf{A}, \{T_e\}_{e\in\mathcal{E}^+}, f\rangle$ for program P^+, the transfer functions T_e for $e \in \mathcal{E}^+ \setminus \mathcal{E}$ being such that for all $\langle \bar{l}_1, \bar{l}_2\rangle \in \mathcal{E}^+$, with $\bar{l}_i \in \{l_i, l_i^+\}$, $T_{\langle l_1, l_2\rangle} = T_{\langle \bar{l}_1, \bar{l}_2\rangle}$.

Proposition 3.16. *Assume the certificates of Fig. 3.6 exist for every* $a_1, a_2, b_1, b_2 \in A$*. Then every certified solution* $\langle S, \vec{c}\rangle$ *for* P *can be transformed into a certified solution* $\langle S^+, \vec{c}'\rangle$ *for* P^+*, s.t.* $S^+(l^+) = S^+(l) = S(l)$ *for all* $l \in \mathsf{dom}(S)$*.*

Proof. We proceed by induction, using the principle derived from the fact that annot *is a sufficient annotation. More concretely, one can attach to every node a weight that corresponds to the length of the longest path to an annotated node, i.e., a node* $l \in \mathsf{dom}(\mathsf{annot})$*.*

For all $l \in \mathcal{N}$ *and* $\bar{l} \in \mathcal{N} \cup \mathcal{N}^+$ *s.t.* $\bar{l} \in \{l, l^+\}$ *we provide the certificates*

- $\mathsf{goal}(l, \bar{l}) :\vdash \overline{\mathsf{annot}}(l) \sqsubseteq \overline{\mathsf{annot}}^+(\bar{l})$*, if* $f = \uparrow$*, or*

- $\mathsf{goal}(l, \bar{l}) :\vdash \overline{\mathsf{annot}}^{+}(\bar{l}) \sqsubseteq \overline{\mathsf{annot}}(l)$, *if* $f = \downarrow$.

For $l, \bar{l}$ *s.t.* $l \in \mathsf{dom}(\mathsf{annot})$, *the certificate* goal *is trivial by definition of* $\overline{\mathsf{annot}}^{+}$, *i.e., an application of the* axiom *operation of the proof algebra. A sketch of the inductive step for the backward case follows, where we implicitly use the condition* $T_{\langle l,l'\rangle} = T_{\langle \bar{l},\bar{l}'\rangle}$. *From the inductive hypotheses* $\mathsf{goal}(l', \bar{l}')$ *for every* $\langle l, l'\rangle \in \mathcal{E}$ *and* $\langle \bar{l}, \bar{l}'\rangle \in \mathcal{E}$ *with* $\bar{l} \in \{l, l^{+}\}$ *and* $\bar{l}' \in \{l', l'^{+}\}$ *we define* $\mathsf{goal}(l, \bar{l})$ *as follows:*

$$
\begin{aligned}
p(l', \bar{l}') &:= \mathsf{goal}(l', \bar{l}') :\vdash \overline{\mathsf{annot}}(l') \sqsubseteq \overline{\mathsf{annot}}^{+}(\bar{l}') \\
q(l', \bar{l}') &:= \mathsf{monot}_T(p_{(l', \bar{l}')}) :\vdash T_{\langle l,l'\rangle}(\overline{\mathsf{annot}}(l')) \sqsubseteq T_{\langle \bar{l},\bar{l}'\rangle}(\overline{\mathsf{annot}}^{+}(\bar{l}')) \\
r(\bar{l}') &:= \mathsf{weak}_{\sqcap}(\{q(l', \bar{l}')\}_{\langle l,l'\rangle \in \mathcal{E}}) \\
&\qquad :\vdash \textstyle\bigsqcap_{\langle l,l'\rangle \in \mathcal{E}} T_{\langle l,l'\rangle}(\overline{\mathsf{annot}}(l')) \sqsubseteq T_{\langle \bar{l},\bar{l}'\rangle}(\overline{\mathsf{annot}}^{+}(\bar{l}')) \\
\mathsf{goal}(l, \bar{l}) &:= \mathsf{intro}_{\sqcap}(\{r(\bar{l}')\}_{\langle \bar{l},\bar{l}'\rangle \in \mathcal{E}^{+}}) : \\
&\qquad :\vdash \textstyle\bigsqcap_{\langle l,l'\rangle \in \mathcal{E}} T_{\langle l,l'\rangle}(\overline{\mathsf{annot}}(l')) \sqsubseteq \bigsqcap_{\langle \bar{l},\bar{l}'\rangle \in \mathcal{E}^{+}} T_{\langle \bar{l},\bar{l}'\rangle}(\overline{\mathsf{annot}}^{+}(\bar{l}'))
\end{aligned}
$$

where, for any sequence $S = \{c_1, c_2, \ldots, c_k\}$, *the expression* $\mathsf{weak}_{\sqcap}(S)$ *stands for* $\mathsf{weak}_{\sqcap}(c_1, \mathsf{weak}_{\sqcap}(\{c_2, \ldots, c_k\}))$, *and similarly with* $\mathsf{intro}_{\sqcap}(S)$. *The inductive step for the forwards case is similar.*

$$
\begin{aligned}
&\mathsf{monot}_T : \mathcal{C}(\vdash a_1 \sqsubseteq a_2) \to \mathcal{C}(\vdash T(a_1) \sqsubseteq T(a_2)) \\
&\mathsf{distr}^{\leftarrow}_{(T,\sqcap)} :\vdash T(a_1) \sqcap T(a_2) \sqsubseteq T(a_1 \sqcap a_2) \\
&\mathsf{distr}^{\rightarrow}_{(T,\sqcap)} :\vdash T(a_1 \sqcap a_2) \sqsubseteq T(a_1) \sqcap T(a_2) \\
&\mathsf{assoc}^{\leftarrow}_{\sqcap} : \mathcal{C}(\vdash a_1 \sqcap (b_1 \sqcap b_2) \sqsubseteq (a_1 \sqcap b_1) \sqcap b_2) \\
&\mathsf{assoc}^{\rightarrow}_{\sqcap} : \mathcal{C}(\vdash (a_1 \sqcap b_1) \sqcap b_2 \sqsubseteq a_1 \sqcap (b_1 \sqcap b_2)) \\
&\mathsf{commut}_{\sqcap} : \mathcal{C}(\vdash a_1 \sqcap a_2 \sqsubseteq a_2 \sqcap a_1)
\end{aligned}
$$

Fig. 3.6. Requirements for certificate translation.

Example 3.4.1 *Figure 3.8 shows the result of applying loop peeling. In the graph, nodes* l_2, l_3, l_4 *and* l_5 *are duplicated into the nodes* l'_2, l'_3, l'_4, *and* l'_5, *respectively, and a new subset of edges is defined accordingly. A certified labeling* $\langle \mathsf{annot}^{+}, \vec{c}^{+}\rangle$, *where* $\mathsf{annot}^{+}(l'_2) = \mathsf{annot}(l_2)$, *is generated for the program in Figure 3.8, by application of Proposition 3.16.*

3.4.2 Edge Transformation

In this section, we consider a basic, but essential, program transformation that represents the effect of replacing statements when the result of the analysis ensures that the semantics is preserved. This is the case, for instance,

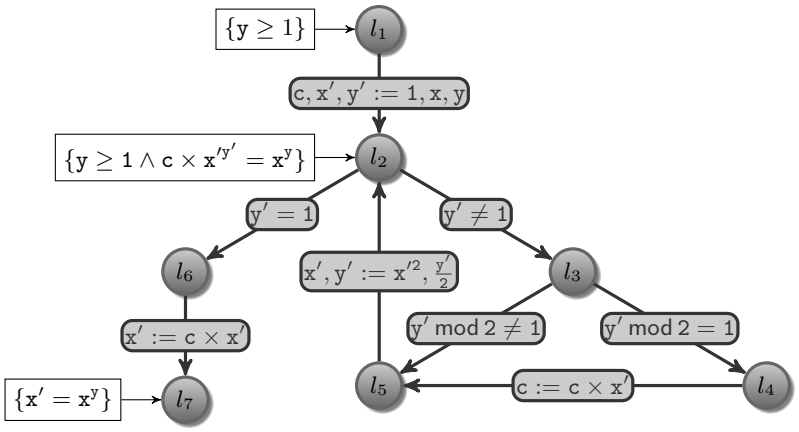


Fig. 3.7. Annotated Program

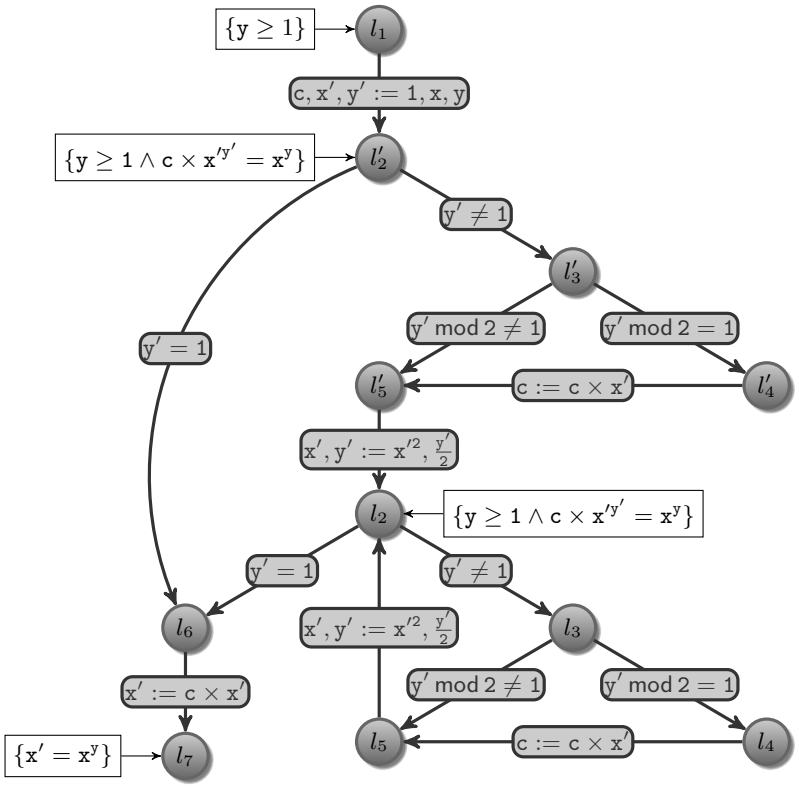


Fig. 3.8. Original Program after Loop Peeling

for constant propagation, in which the analysis infers whether some variable invariably holds a constant value in certain program points. In a second step, constant propagation optimizes statements assuming the validity of the analysis results. Consider the following fragment of code:

$$\begin{array}{l} l_1 : \texttt{x} := 0; \\ l_2 : \texttt{if } b \texttt{ then } y := y + x; x := x + 1; \\ l_3 : \texttt{while } b \texttt{ do } y := y + x; x := x + 1 \quad . \end{array}$$

Assume a static analysis infers that, for every program execution, x holds the value 0 at the program point l_2. Assuming this, the first occurrence of the statement $y := y + x; x := x + 1$ can be replaced by the semantically equivalent statement $\texttt{skip}; x := 1$. This program transformation is reflected in the abstract program representation by a transformation on the relation associated to each edge. Since the semantics relation is modified, we must also assume that the corresponding abstract interpretation transfer functions are modified in accordance. Furthermore, we also consider the case in which some of the original edges are removed.

In the rest of this section, we formally describe the program transformation under consideration, and we explain and define when a labeling representing an analysis result justifies a program transformation. Then, we state the existence of certificate translators based on the certifiability of the labeling that justifies the transformation.

Let P be a program $\langle \mathcal{N}, \mathcal{E}, \rightsquigarrow, l_{\mathsf{init}} \rangle$, the program $P' = \langle \mathcal{N}', \mathcal{E}', \rightsquigarrow', l_{\mathsf{init}} \rangle$ is transformed from P if P' is a subgraph of P such that $\mathcal{N}' \subseteq \mathcal{N}$ and $\mathcal{E}' \subseteq \mathcal{E}$. Consider the two abstract interpretations $I = \langle \mathbf{A}, \{T_e\}_{e \in \mathcal{E}}, f \rangle$ and $I' = \langle \mathbf{A}, \{T'_e\}_{e \in \mathcal{E}'}, f \rangle$ for P and P', respectively. Note that the abstract interpretations only differ in the set of transfer functions.

We say that the transformation is justified by the result of the analysis if its validity implies an equivalence between the original and transformed semantics relation. That is, for every edge $\langle l, l' \rangle$, and $\eta, \eta' \in \mathsf{Env}$ s.t. $\eta \models S(l)$, we have that $\langle l, \eta \rangle \rightsquigarrow \langle l', \eta' \rangle$ iff $\langle l, \eta \rangle \rightsquigarrow' \langle l', \eta' \rangle$. However, for the purpose of certificate translation we need a stronger property than semantics preservation. The following result states that it is possible to translate the certificates as long as there exists a certificate justif stating the equivalence of the corresponding transfer functions.We assume that the analyzer phase is certifying, i.e., we require the analysis result to be represented and certified in the verification environment. From Proposition 3.14, the existence of such a certified solution follows from the certified consistency (i.e., from the existence of the certificate cons defined in Section 3.3) of the abstract interpretation representing the analysis w.r.t. the more concrete abstract interpretation representing the verification environment.

Proposition 3.17 (Existence of certificate translators). *Let $\langle S, \vec{c}^S \rangle$ be a certified solution for I such that for every $\langle l_1, l_2 \rangle \in \mathcal{E}'$ and $a \in A$:*

- *if $f = \uparrow$ then $\mathsf{justif}(l_1, l_2) :\vdash S(l_1) \sqcap T_{\langle l_1, l_2 \rangle}(a) \sqsubseteq T'_{\langle l_1, l_2 \rangle}(a)$;*

- *if* $f = \downarrow$ *then* $\mathsf{justif}(l_1, l_2) :\vdash T'_{\langle l_1,l_2\rangle}(a) \sqsubseteq S(l_2) \sqcap T_{\langle l_1,l_2\rangle}(a)$

Then, provided the certificates in Figure 3.6 are given for every $a_1, a_2, b_1, b_2 \in A$*, one can transform every certified labeling* $\langle \mathsf{annot}, \vec{c}\rangle$ *for* P *into a certified labeling* $\langle \mathsf{annot}', \vec{c}'\rangle$ *for* P'*, where* $\mathsf{annot}'(l)$ *is defined as* $\mathsf{annot}(l) \sqcap S(l)$ *for every node* l *in* $\mathsf{dom}(\mathsf{annot}') = \mathsf{dom}(\mathsf{annot}) \cap \mathcal{N}'$*.*

Proof. We build for every l *in* $\mathcal{N}'$ *the certificate*

- $\mathsf{goal}(l) :\vdash S(l) \sqcap \overline{\mathsf{annot}}(l) \sqsubseteq \overline{\mathsf{annot}}'(l)$ *if* $f = \uparrow$*, or*
- $\mathsf{goal}(l) :\vdash \overline{\mathsf{annot}}'(l) \sqsubseteq S(l) \sqcap \overline{\mathsf{annot}}(l)$ *if* $f = \downarrow$*,*

from which the existence of a certificate for annot' *follows.*

We proceed by induction, using the principle derived from the fact that annot *is a sufficient annotation. More concretely, one can attach to every node a weight that corresponds to the length of the longest path to an annotated node, i.e., a node* $l \in \mathsf{dom}(\mathsf{annot})$*. In the base case, where* $l \in \mathsf{dom}(\mathsf{annot}')$*, the certificate* $\mathsf{goal}(l)$ *is defined trivially, since* $\overline{\mathsf{annot}}'(l) = S(l) \sqcap \overline{\mathsf{annot}}(l)$*. For the inductive step, where* $l \notin \mathsf{dom}(\mathsf{annot}')$*, the proof is given in Figures 3.9 and 3.10 for the backward and forward case, respectively, where the application of certificates* $\mathsf{assoc}^{\leftarrow}_{\sqcap}$*,* $\mathsf{assoc}^{\rightarrow}_{\sqcap}$*, and* $\mathsf{commut}_{\sqcap}$ *is omitted for readability.*

Let $a = S(l)$, $a' = S(l')$, $T = T_{\langle l,l'\rangle}$ and $T' = T'_{\langle l,l'\rangle}$ in:

$\mathsf{hyp}_1 := \mathsf{monot}_T : \mathcal{C}(\vdash b_1 \sqsubseteq b_2) \to \mathcal{C}(\vdash T(b_1) \sqsubseteq T(b_2))$
$\mathsf{hyp}_2 := \mathsf{distrib}_T : \mathcal{C}(\vdash T(b_1) \sqcap T(b_2) \sqsubseteq T(b_1 \sqcap b_2))$
$p_1 := \mathsf{goal}(l') :\vdash a' \sqcap \overline{\mathsf{annot}}(l') \sqsubseteq \overline{\mathsf{annot}}'(l')$
$p_2 := \mathsf{hyp}_1(p_1) :\vdash T'(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l'))$
$p_3 := \mathsf{justif}(l, l') :\vdash a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T'(a' \sqcap \overline{\mathsf{annot}}(l'))$
$p_5 := \mathsf{elim}_{\sqcap}(\mathsf{weak}_{\sqcap}(-, p_2), p_3) :\vdash a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l'))$
$p_6 := \mathsf{hyp}_2 :\vdash T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq T(a' \sqcap \overline{\mathsf{annot}}(l'))$
$p_7 := \mathsf{axiom} :\vdash a \sqsubseteq a$
$p_8 := \mathsf{weak}_{\sqcap}(p_7) :\vdash a \sqcap T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq a$
$p_9 := \mathsf{weak}_{\sqcap}(p_6) :\vdash a \sqcap T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq T(a' \sqcap \overline{\mathsf{annot}}(l'))$
$p_{10} := \mathsf{intro}_{\sqcap}(p_8, p_9) :\vdash a \sqcap T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l'))$
$p_{11} := \mathsf{elim}_{\sqcap}(\mathsf{weak}_{\sqcap}(p_5), p_{10}) :\vdash a \sqcap T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l'))$
$p_{12} := \vec{c}_l^{\,S} :\vdash a \sqsubseteq T(a')$
$p_{13} := \mathsf{elim}_{\sqcap}(p_{11}, p_{12}) :\vdash a \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l'))$
$p_{14} := \mathsf{weak}_{\sqcap}(p_{13}) :\vdash a \sqcap \bigsqcap_{\langle l,l'\rangle \in \mathcal{E}} T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l'))$
$\mathsf{goal}(l) := \mathsf{intro}_{\sqcap}(\{p_{12}\}_{\langle l,l'\rangle \in \mathcal{E}}) :$
$\vdash a \sqcap \bigsqcap_{\langle l,l'\rangle \in \mathcal{E}} T(\overline{\mathsf{annot}}(l')) \sqsubseteq \bigsqcap_{\langle l,l'\rangle \in \mathcal{E}} T'(\overline{\mathsf{annot}}'(l'))$

Fig. 3.9. Definition of $\mathsf{goal}(l)$ for certificate translation (case $f = \uparrow$)

Let $a = S(l)$, $a' = S(l')$, $T = T_{\langle l',l\rangle}$ and $T' = T'_{\langle l',l\rangle}$ in:

$$
\begin{aligned}
p_1 &:= \mathsf{goal}'(l') : \vdash \overline{\mathsf{annot}}'(l') \sqsubseteq a' \sqcap \overline{\mathsf{annot}}(l') \\
p_2 &:= \mathsf{monot}_{T'} : \vdash T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq T'(a' \sqcap \overline{\mathsf{annot}}(l')) \\
p_3 &:= \mathsf{justif} : \vdash T'(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l')) \\
p_4 &:= \mathsf{distrib}_T : \vdash T(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \\
p_5 &:= \mathsf{weak}_{\sqcap}(p_4) : \vdash a \sqcap T(a' \sqcap \overline{\mathsf{annot}}(l')) \sqsubseteq T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \\
p_6 &:= \vec{c}^{S}_{\langle l',l\rangle} : \vdash T(a') \sqsubseteq a \\
p_7 &:= \mathsf{weak}_{\sqcap}(p_6) : \vdash T(a') \sqcap T(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \\
p_8 &:= \mathsf{weak}_{\sqcap}(\mathsf{axiom}) : \vdash T(a') \sqcap T(\overline{\mathsf{annot}}'(l')) \sqsubseteq T(\overline{\mathsf{annot}}'(l')) \\
p_9 &:= \mathsf{intro}_{\sqcap}(p_7, p_8) : \vdash T(a') \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq a \sqcap T(\overline{\mathsf{annot}}(l')) \\
p_{10} &:= \mathsf{trans}(p_2, \mathsf{trans}(p_3, \mathsf{trans}(p_5,))) : \\
&\qquad\qquad T'(\overline{\mathsf{annot}}'(l'))a \sqcap T(\overline{\mathsf{annot}}(l')) \\
p_{11} &:= \mathsf{weak}_{\sqcup}(\mathsf{axiom}) : \vdash T(\overline{\mathsf{annot}}(l')) \sqsubseteq \textstyle\bigsqcup_{\langle l,l'\rangle\in\mathcal{E}} T(\overline{\mathsf{annot}}(l')) \\
p_{12} &:= \mathsf{intro}_{\sqcap}(\mathsf{weak}_{\sqcap}(\mathsf{axiom}), \mathsf{weak}_{\sqcap}(p_{11})) : \\
&\qquad\qquad \vdash a \sqcap T(\overline{\mathsf{annot}}(l')) \sqsubseteq a \sqcap \textstyle\bigsqcup_{\langle l,l'\rangle\in\mathcal{E}} T(\overline{\mathsf{annot}}(l')) \\
p_{13} &:= \mathsf{trans}(p_{10}, p_{12}) : \vdash T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \sqcap \textstyle\bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T(\overline{\mathsf{annot}}(l')) \\
\mathsf{goal}(l) &:= \mathsf{intro}_{\sqcup}(\{p_{13}\}_{\langle l',l\rangle\in\mathcal{E}'}) : \\
&\qquad\qquad \vdash \textstyle\bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T'_{\langle l',l\rangle}(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T(\overline{\mathsf{annot}}(l'))
\end{aligned}
$$

Fig. 3.10. Definition of $\mathsf{goal}(l)$ for certificate translation (case $f = \downarrow$)

Using the results of Proposition 3.14, Proposition 3.17 can be instantiated to prove the existence of certificate transformers for many common optimizations, including constant propagation and common sub-expression elimination. In a nutshell, one first runs the certifying analyzer, which provides the solution S, then performs the optimization, and finally one provides a justification $\mathsf{justif}(l_1, l_2)$ for each edge (instruction) that has been modified by the optimization. This process is further illustrated in the following example.

Example 3.4.2 *Suppose that we know (e.g., from the execution context) that the program is called with an even y; such knowledge is formalized by a precondition $y = 2 \times p$. Then, one can consider a forward abstract interpretation that analyses parity of variables and which variables are modified. A certifying analyzer for such an abstract interpretation exists by Proposition 3.14 and will produce a certified solution $\langle S, \vec{c}^{S}\rangle$ such that S associates the assertion $y = 2 \times p$ to the node l_1, the assertion $y' = 2 \times p \wedge x = x'$ to the nodes $\{l'_2, l'_3, l'_5\}$, and* true *to any other node. Figure 3.11 shows a partial labeling with the information computed by the analysis.*

Figure 3.12 contains an optimized version of the program of Figure 3.11, where jump statements whose conditions can be determined statically have been eliminated (nodes l'_2 and l'_3) and unreachable nodes have been removed (node l'_4), and where assignments have been simplified by propagating the results of the analysis (node l'_5). By Proposition 3.17, one can build a certificate for the optimized program, with labeling $\mathsf{annot}'(l) = \mathsf{annot}(l) \sqcap S(l)$ for all nodes

$l \in \mathsf{dom}(\mathsf{annot})$, *provided there exists, for every* $a \in A$ *and for every modified edge, i.e., for every* $\langle l, l' \rangle \in \{\langle l'_2, l'_3 \rangle, \langle l'_3, l'_5 \rangle, \langle l'_5, l_2 \rangle\}$, *a certificate:*

$$\mathsf{justif}_{\langle l,l' \rangle} :\vdash y' = 2 \times p \wedge x = x' \sqcap T_{\langle l,l' \rangle}(a) \sqsubseteq T'_{\langle l,l' \rangle}(a)$$

The remaining certificates $\mathsf{justif}(l, l')$ *for* $\langle l, l' \rangle \notin \{\langle l'_2, l'_3 \rangle, \langle l'_3, l'_5 \rangle, \langle l'_5, l_2 \rangle\}$ *are trivially generated since* $T'_{\langle l,l' \rangle} = T_{\langle l,l' \rangle}$.

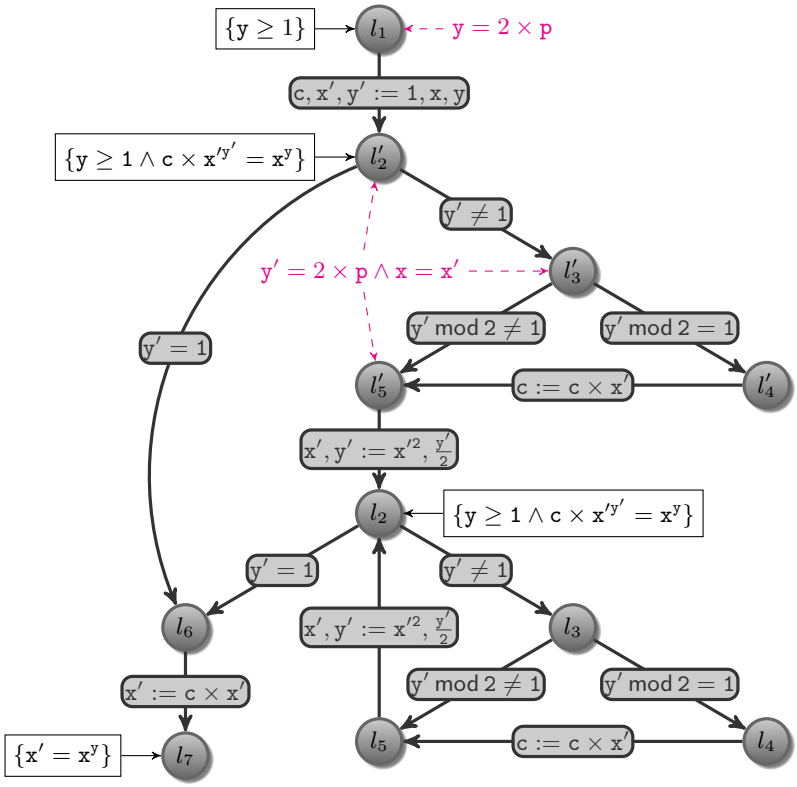


Fig. 3.11. Program Analysis after Loop Peeling

3.4.3 Program Representation Abstraction

Proposition 3.17 requires that the transformation is justified for each edge of the program; this rules out several well known optimizations such as instruction swapping or code motion, whose justification involves more than one instruction. Consider for example the sequential composition $c_1; c_2$, where c_1 and c_2 are atomic statements. Assume that the statement $c_1; c_2$ is represented

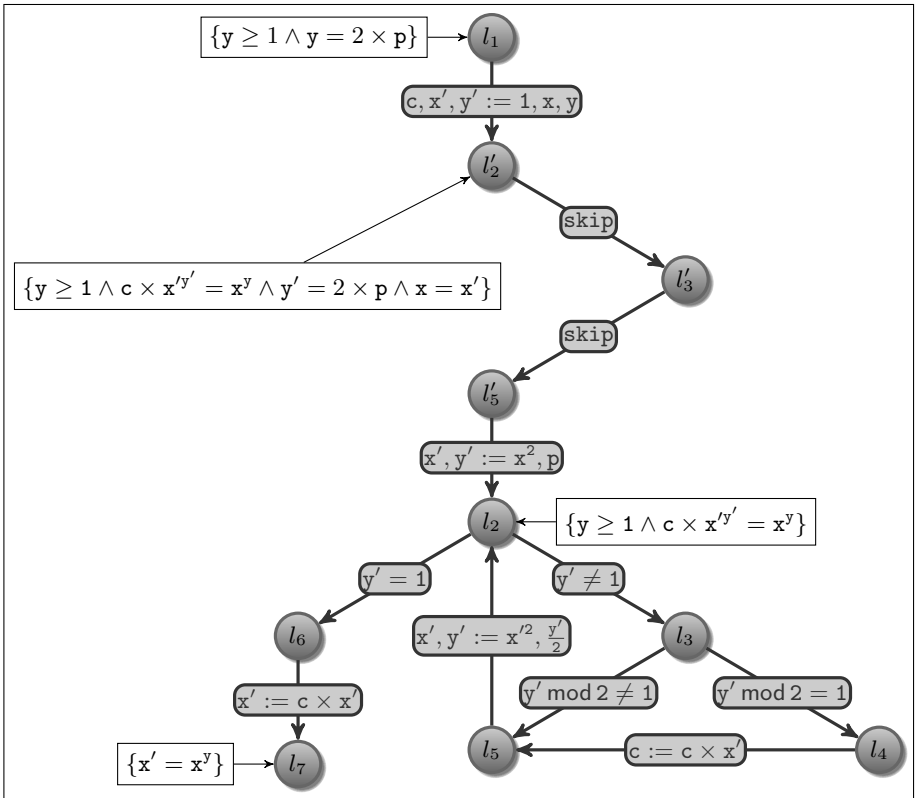


Fig. 3.12. Program after Optimizing Transformations

as a three node graph structure, with $\mathcal{N} = \{l_1, l_2, l_3\}$ and $\mathcal{E} = \{\langle l_1, l_2\rangle, \langle l_2, l_3\rangle\}$ such that the edge $\langle l_1, l_2\rangle$ represents the execution of c_1 and $\langle l_2, l_3\rangle$ represents the execution of c_2. Suppose that c_1 and c_2 are independent, that is, that the order in which they are executed does not alter the final state. If each edge represents one instruction, and if we model the transformation as described in Section 3.4.2, we have that transfer functions $T_{\langle l_1,l_2\rangle}$ and $T_{\langle l_2,l_3\rangle}$ are replaced by $T_{\langle l_2,l_3\rangle}$ and $T_{\langle l_1,l_2\rangle}$, respectively. Then, to prove the existence of a certificate translator, from Proposition 3.17, since we assume the labeling S representing the analysis is s.t. $S(l) = \top$ for $l \in \mathcal{N}$, we must show that $T_{\langle l_1,l_2\rangle} = T_{\langle l_2,l_3\rangle}$, which does not necessarily hold. To overcome this limitation, one must abandon the intuitive representation of programs, where each edge represents one instruction, and cluster several instructions into a single edge.

In this section, we formally capture this idea of clustering, and use it to extend the applicability of the results of Section 3.4.2. First we define an abstraction of the program model, by selecting a set of distinguished nodes $\mathcal{N}_0 \subseteq \mathcal{N}$. The new representation preserves the program structure, but abstracts away

the program points represented by the nodes $\mathcal{N}_1 = \mathcal{N} \setminus \mathcal{N}_0$. Consequently, a coarser grained semantics relation must be defined in replacement of the original one. That is done later in this section by taking the transitive closure over the execution states with nodes in $\mathcal{N}_1$.

Let $P = \langle \mathcal{N}, \mathcal{E}, \rightsquigarrow, l_{\mathsf{init}} \rangle$ be a program and $\mathcal{N}_0 \cup \mathcal{N}_1 = \mathcal{N}$ a partition of nodes such that $\mathcal{E} \cap (\mathcal{N}_1 \times \mathcal{N}_1)$ does not contain infinite chains (i.e., there are no loops in $\mathcal{E} \cap (\mathcal{N}_1 \times \mathcal{N}_1)$). The set of edges relating nodes in the frontier between $\mathcal{N}_0$ and $\mathcal{N}_1$ are defined by the relations $\mathcal{E}_{0\times1} = \mathcal{E} \cap (\mathcal{N}_0 \times \mathcal{N}_1)$ and $\mathcal{E}_{1\times0} = \mathcal{E} \cap (\mathcal{N}_1 \times \mathcal{N}_0)$. The relation $\mathcal{E}_{1\times1}$ denotes the restriction of $\mathcal{E}$ on the domain $\mathcal{N}_1$, i.e., $\mathcal{E}_{1\times1} = \mathcal{E} \cap (\mathcal{N}_1 \times \mathcal{N}_1)$ (and similarly with $\mathcal{E}_{0\times0}$). Finally, the relation $\hat{\mathcal{E}}$ is defined as $\mathcal{E}_{0\times0} \cup (\mathcal{E}_{0\times1} \circ (\mathcal{E}_{1\times1})^\star \circ \mathcal{E}_{1\times0})$, where $R^\star$ denotes the reflexive and transitive closure of a relation R, and $x(R_1 \circ R_2)y$ is defined as $\exists z.\ (xR_1z) \wedge (zR_2y)$, for any relations R_1 and R_2.

Assume that $l_{\mathsf{init}} \in \mathcal{N}_0$. Let $\hat{\rightsquigarrow}$ be defined as $\rightsquigarrow_{0\times0} \cup (\rightsquigarrow_{0\times1} \circ (\rightsquigarrow_{1\times1})^\star \circ \rightsquigarrow_{1\times0})$, where $\rightsquigarrow_{i\times j}$ is defined as $\rightsquigarrow \cap ((\mathcal{N}_i \times \mathsf{Env}) \times (\mathcal{N}_j \times \mathsf{Env}))$. Then $\hat{P} = \langle \mathcal{N}_0, \hat{\mathcal{E}}, \hat{\rightsquigarrow}, l_{\mathsf{init}} \rangle$ is a program s.t. for every $l, l' \in \mathcal{N}_0$, $\langle l, \eta \rangle \hat{\rightsquigarrow}^\star \langle l', \eta' \rangle$ iff $\langle l, \eta \rangle \rightsquigarrow^\star \langle l', \eta' \rangle$.

Let $\vec{\mathcal{N}}$ denote the set of program paths, i.e., the sequences $\vec{ls}$ of elements in $\mathcal{N}$ such that for every consecutive elements l_i, l_j in $\vec{l}$, $\langle l_i, l_j \rangle \in \mathcal{E}$.

For $\vec{ls} \in \vec{\mathcal{N}}$, we define $\check{T}_{\vec{ls}}$ inductively on the length of the program path $\vec{ls}$ as a composition of the transfer functions corresponding to the edges in $\vec{ls}$:

$$\check{T}_{\langle l,l' \rangle} = T_{\langle l,l' \rangle}$$
$$\check{T}_{\langle l_1,\ldots,l_k,l \rangle} = \begin{cases} T_{\langle l_k,l \rangle} \circ \check{T}_{\langle l_1,\ldots,l_k \rangle} \text{ if } f = \downarrow \\ \check{T}_{\langle l_1,\ldots,l_k \rangle} \circ T_{\langle l_k,l \rangle} \text{ if } f = \uparrow \end{cases}$$

Finally, we define $\hat{T}_{\langle l,l' \rangle}(a)$ as

- $\bigsqcup \{ \check{T}_{\langle l,\vec{ls},l' \rangle}(a) \mid \vec{ls} \in (\mathcal{N}_0 \times \mathcal{N}_1^\star \times \mathcal{N}_0) \cap \vec{\mathcal{N}} \} \cup \{ T_{\langle l,l' \rangle} \mid \langle l, l' \rangle \in \mathcal{E} \}$ if $f = \downarrow$; and
- $\bigsqcap \{ \check{T}_{\langle l,\vec{ls},l' \rangle}(a) \mid \vec{ls} \in (\mathcal{N}_0 \times \mathcal{N}_1^\star \times \mathcal{N}_0) \cap \vec{\mathcal{N}} \} \cup \{ T_{\langle l,l' \rangle} \mid \langle l, l' \rangle \in \mathcal{E} \}$ if $f = \uparrow$.

It follows from the definition of $\hat{I}$, that if I is consistent with the semantics of P then $\hat{I}$ is consistent with the semantics of $\hat{P}$.

Lemma 3.18. *Let S s.t. $\mathsf{dom}(S) \subseteq \mathcal{N}_0$. Then $\langle S, \vec{c} \rangle$ is a certified solution for I iff $\langle S, \vec{c} \rangle$ is a certified solution of $\hat{I} = \langle \mathbf{A}, \hat{T}_e, f \rangle$.*

The results of Section 3.4.2 are immediately extended to a broader set of proof transformations:

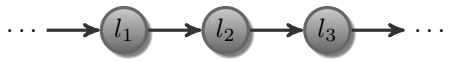
Corollary 3.19. *Let $\langle S, \vec{c} \rangle$ be a certified solution of I, with $\mathsf{dom}(S) \subseteq \mathcal{N}_0$. Suppose that $\mathcal{N}_0' \subseteq \mathcal{N}_0$ and $\hat{\mathcal{E}}' \subseteq \hat{\mathcal{E}}$ and for every $\langle l_1, l_2 \rangle \in \hat{\mathcal{E}}'$ and $a \in A$:*

- *if $f = \uparrow$ then $\mathsf{justif}(l_1, l_2) :\vdash S(l_1) \sqcap \hat{T}_{\langle l_1,l_2 \rangle}(a) \sqsubseteq \hat{T}'_{\langle l_1,l_2 \rangle}(a)$;*
- *if $f = \downarrow$ then $\mathsf{justif}(l_1, l_2) :\vdash \hat{T}'_{\langle l_1,l_2 \rangle}(a) \sqsubseteq S(l_2) \sqcap \hat{T}_{\langle l_1,l_2 \rangle}(a)$*

Then every certified labeling $\langle \mathsf{annot}, \vec{c} \rangle$ *for* P *such that* $\mathsf{dom}(\mathsf{annot}) \subseteq \mathcal{N}_0$ *can be transformed into a certified labeling* $\langle \mathsf{annot}', \vec{c}' \rangle$ *for* P'*, where* $\mathsf{annot}'(l)$ *is defined as* $\mathsf{annot}(l) \sqcap S(l)$ *for all* $l \in \mathsf{dom}(\mathsf{annot}') = \mathsf{dom}(\mathsf{annot}) \cap \mathcal{N}'$.

Proof. Notice that $\hat{P}$ *is a program as defined in Definition 3.1. From Proposition 3.18),* $\langle S, \vec{c} \rangle$ *is a certified solution of* $\hat{I}$ *and* $\langle \mathsf{annot}, \vec{c} \rangle$ *is a certified labeling for* $\hat{P}$*. The corollary follows from the existence of the certificate* justif *and Lemma 3.17.*

Corollary 3.19 restates the result of Proposition 3.17, but requiring the certificate justif to be defined for every edge in $\hat{\mathcal{E}}$ instead of every edge in $\mathcal{E}$. Therefore, it is a generalization that covers a wider range of program transformations. Consider for instance the case of instruction swapping, represented in the program graph



by swapping the semantics of the edges $\langle l_1, l_2 \rangle$ and $\langle l_2, l_3 \rangle$. In this case, the structure of the graph is preserved by the transformation, i.e., $\mathcal{N}' = \mathcal{N}$ and $\mathcal{E}' = \mathcal{E}$, but transfer functions are such that $T'_{\langle l_1, l_2 \rangle} = T_{\langle l_2, l_3 \rangle}$ and $T_{\langle l_2, l_3 \rangle} = T_{\langle l_1, l_2 \rangle}$. In the previous scenario, we cannot a priori show a correspondence between T_e and T'_e for any $e \in \{\langle l_1, l_2 \rangle, \langle l_2, l_3 \rangle\}$. However, by clustering nodes, we are in a position to analyze whether $T'_{\langle l_1, l_2 \rangle} \circ T'_{\langle l_2, l_3 \rangle} = T_{\langle l_1, l_2 \rangle} \circ T_{\langle l_2, l_3 \rangle}$, i.e., $\hat{T}'_{\langle l_1, l_3 \rangle} = \hat{T}_{\langle l_1, l_3 \rangle}$.

Example 3.4.3 *We illustrate how to apply the result of this section to the program of Figure 3.12 to obtain the program of Figure 3.13. The program in the figure shows that, after a program optimization, the edges* $\langle l'_2, l'_3 \rangle$ *and* $\langle l'_3, l'_5 \rangle$ *represent program transitions that do not modify the execution state. That is modeled in the analysis domain by defining the transfer functions* $T_{\langle l'_2, l'_3 \rangle}$ *and* $T_{\langle l'_3, l'_5 \rangle}$ *as the identity function in* A*. Let* P' *be the result of removing the nodes* l'_3 *and* l'_5*, and replacing the edges* $\langle l'_2, l'_3 \rangle$*,* $\langle l'_3, l'_5 \rangle$ *and* $\langle l'_5, l_2 \rangle$ *by a single edge* $\langle l'_2, l_2 \rangle$*. The abstract interpretation* I' *for program* P' *is s.t.* $T_{\langle l'_2, l_2 \rangle}$ *is defined as* $T_{\langle l'_2, l'_3 \rangle} \circ T_{\langle l'_3, l'_5 \rangle} \circ T_{\langle l'_5, l_2 \rangle}$ *if* $f = \uparrow$ *and* $T_{\langle l'_5, l_2 \rangle} \circ T_{\langle l'_3, l'_5 \rangle} \circ T_{\langle l'_2, l'_3 \rangle}$ *if* $f = \downarrow$.

However, this transformation is not considered a certificate translation as defined in Section 3.4.2, since it is not a single-edge by single-edge replacement.

If we define the set $\mathcal{N}_1$ *as* $\{l'_3, l'_5\}$ *, we have that* $\hat{\mathcal{E}} = \hat{\mathcal{E}}'$ *and* $\hat{T}_e = \hat{T}'_e$ *for every* $e \in \hat{\mathcal{E}}$*. Therefore, it is straightforward to apply the results of Corollary 3.19, with* S *defined as* true *for every annotated program node (i.e.,* $\{l_1, l'_2, l_2, l_7\}$*). However, for notational simplicity, we define* $S'(l) = S(l)$ *instead of* $S'(l) = S(l) \wedge \mathsf{true}$ *(*$l \in \mathsf{dom}(S)$*).*

Proof Preserving Compilation

Proposition 3.19 can be specialized to prove preservation of proof obligations for non-optimizing compilers [14, 8]. Indeed, non-optimizing compilation

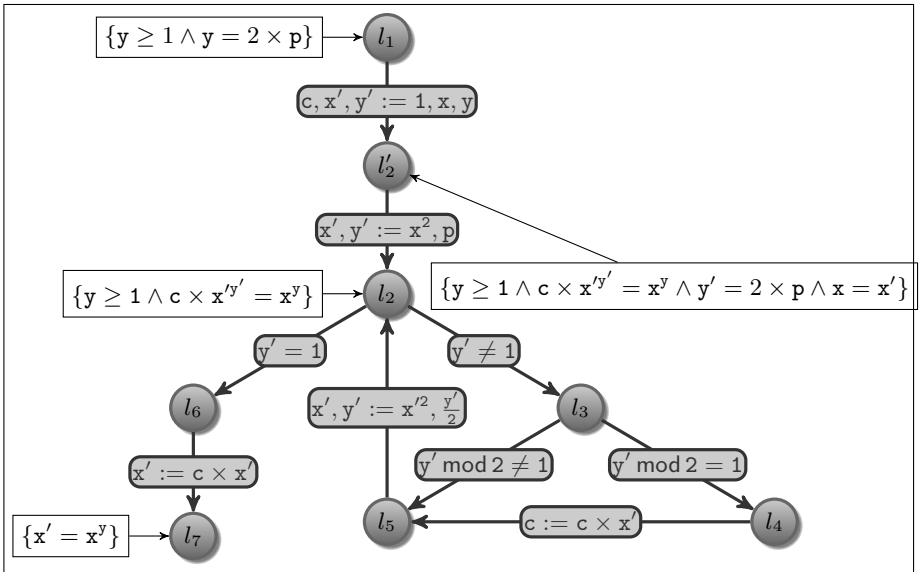


Fig. 3.13. Program after Node Coalescing

transforms a graph representation of a program by splitting each node into a subgraph of more basic nodes, preserving the overall program structure. Thus, one can coalesce back the generated subgraphs into a skeleton structure similar to the source program. In this case the transformation from P to P' needs not be justified by the result of an analysis (formally one chooses S s.t. $S(l) = \top$ for every $l \in \mathcal{N}'$). If we assume that transfer functions of the skeleton representation are equal to those of the source program (it is not sufficient that the functions are equivalent w.r.t. $\sqsubseteq$; equality is essential), then proof obligations are preserved and certificates can be reused without modification.

Instruction Insertion

Consider an imperative language as the one shown in Figure 3.1. A variable is fresh with respect to a program and a specification (i.e., a labeling) if it does not belong to the program nor to the specification. Consider a program transformation consisting in the insertion of instructions that only affect fresh variables. In our abstract model, given a particular abstract domain, the insertion of such instructions is represented as a new edge with a transfer function defined as the identity in such a domain (although there may be a wider abstract domain in which the transfer function is not the identity function).

The following statement together with Corollary 3.19 enables us to consider this simple program transformation.

Proposition 3.20. *Let $I = \langle \mathbf{A}, T_e, f\rangle$ and $I' = \langle \mathbf{A}', T_e, f\rangle$ such that $\mathbf{A}'$ is a sublattice of $\mathbf{A}$ and A' is closed under T_e. Then, $\langle S, \vec{c}\rangle$ s.t. for every $l \in \mathcal{N}$ $S(l) \in A'$, is a certified solution of I iff it is a certified solution of I'.*

Consider for example the abstract interpretation $I = \langle \mathbf{A}, \{T_e\}, \uparrow\rangle$ where $\mathbf{A}$ is the lattice of logical formulae and $\{T_e\}_{e\in\mathcal{E}}$ are defined as weakest precondition transformers. Consider a program $P = \langle \mathcal{N}, \mathcal{E}, \rightsquigarrow, l_{\mathsf{init}}\rangle$ with $\langle l_1, l_2\rangle \in \mathcal{E}$ and $\langle S, \vec{c}\rangle$ a certified solution of I. Let P' be the result of inserting between nodes l_1 and l_2 an extra program point to represent an affectation to a fresh variable x, occurring neither in the program nor in the specification (i.e., labeling S). More precisely, $\mathcal{N}' = \mathcal{N} \cup \{l\}$, $\mathcal{E}' = (\mathcal{E} \setminus \{\langle l_1, l_2\rangle\}) \cup \{\langle l_1, l\rangle, \langle l, l_2\rangle\}$ and $T'_{\langle l_1,l\rangle} = T_{\langle l_1,l_2\rangle}$ and $T'_{\langle l_1,l\rangle}$ defined as an affectation on x. Let $\mathcal{N}_1$ be the singleton $\{l\}$, we still cannot apply Corollary 3.19 since $\hat{T'}_{\langle l_1,l_2\rangle}$ is not necessarily equal to $\hat{T}_{\langle l_1,l_2\rangle}$. Consider instead the abstract interpretations $I_x = \langle \mathbf{A}_x, T_e, \uparrow\rangle$ and $I'_x = \langle \mathbf{A}_x, T'_e, \uparrow\rangle$, s.t. $\mathbf{A}_x$ is the sublattice of logical formulae that does not contain the variable x. Since x is a fresh variable, it does not appear in $S(l)$ for any l, and the transfer functions $\{T_e\}_{e\in\mathcal{E}}$ and $\{T_e\}_{e\in\mathcal{E}}$ are closed on A_x. Then $\langle S, \vec{c}\rangle$ is a certified solution of I_x by Proposition 3.20. Since for every $\varphi \in A_x$, $\hat{T'}_{\langle l_1,l_2\rangle}(\varphi) = \hat{T'}_{\langle l_1,l_2\rangle}(\varphi)$, $\langle S, \vec{c}\rangle$ is a certified solution of I'_x by Corollary 3.19. And again by Proposition 3.20 it is also a certified solution of I'.

3.4.4 Second-Order Analysis-Based Optimizations

Our definition of abstract interpretations targets the analyses of safety properties, and is not general enough to model liveness analysis. In consequence, certificate transformation for transformations such as dead variable elimination is not covered by previous results. In addition, we have only considered in previous sections the cases in which it is sufficient to strengthen annotations to enable certificate translation. In this section, we provide a mild generalization of previous results, in which the result of the analysis is merged with the original annotations with a generic composition operator. As we illustrate with a dead variable elimination example at the end of this section, the composition operator may be instantiated as a weakening of the original annotations.

The following results, generalizes Proposition 3.17 in terms of an arbitrary composition operator $\centerdot : A \times A \to A$.

Proposition 3.21. *Let $\centerdot : A \times A \to A$ be a composition operator s.t. for every $a_1, a_2, b_1, b_2 \in A$ there exists a certificate*

$$\mathsf{monot}_{\centerdot} : \mathcal{C}(\vdash a_1 \sqsubseteq a_2) \to \mathcal{C}(\vdash b_1 \sqsubseteq b_2) \to \mathcal{C}(\vdash a_1 \centerdot b_1 \sqsubseteq a_2 \centerdot b_2)$$

Let $\langle S, \vec{c}^S\rangle$ be a certified solution for I s.t. for every $\langle l_1, l_2\rangle \in \mathcal{E}'$ and $a \in A$:

- *if $f = \uparrow$ then* $\mathsf{justif}(l_1, l_2) : \vdash S(l_1) \centerdot T_{\langle l_1,l_2\rangle}(a) \sqsubseteq T'_{\langle l_1,l_2\rangle}(a \centerdot S(l_2))$;
- *if $f = \downarrow$ then* $\mathsf{justif}(l_1, l_2) : \vdash T'_{\langle l_1,l_2\rangle}(a \centerdot S(l_1)) \sqsubseteq S(l_2) \centerdot T_{\langle l_1,l_2\rangle}(a)$

Then, provided the certificate monot_T *defined in Figure 3.6 exists for all* $a_1, a_2 \in A$*, every certified labeling* $\langle \mathsf{annot}, \vec{c} \rangle$ *for* P *can be transformed into a certified labeling* $\langle \mathsf{annot}', \vec{c}' \rangle$ *for* P'*, where* $\mathsf{annot}'(l) = \mathsf{annot}(l) \centerdot S(l)$ *for every node* l *in* $\mathsf{dom}(\mathsf{annot}') = \mathsf{dom}(\mathsf{annot}) \cap \mathcal{N}'$*.*

Proof. The proof is similar to that of Proposition 3.17. The definition of the certificate goal*, defined such that*

$$\mathsf{goal}(l) :\vdash a \centerdot \textstyle\bigsqcap_{\langle l,l' \rangle \in \mathcal{E}} T_{\langle l,l' \rangle}(\overline{\mathsf{annot}}(l')) \sqsubseteq \bigsqcap_{\langle l,l' \rangle \in \mathcal{E}} T'_{\langle l,l' \rangle}(\overline{\mathsf{annot}}'(l')) \text{ if } f = \uparrow$$
$$\mathsf{goal}(l) :\vdash \textstyle\bigsqcup_{\langle l',l \rangle \in \mathcal{E}} T'_{\langle l',l \rangle}(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \centerdot \bigsqcup_{\langle l',l \rangle \in \mathcal{E}} T_{\langle l',l \rangle}(\overline{\mathsf{annot}}(l')) \text{ if } f = \downarrow$$

and from which the existence of $\vec{c}'$ *follows, is sketched in Figures 3.14 and 3.15.*

Let $a = \gamma(S(l))$, $a' = \gamma(S(l'))$, $T = T_{\langle l,l' \rangle}$ and $T' = T'_{\langle l,l' \rangle}$ in:

$$\begin{aligned}
p_1 &:= \mathsf{goal}(l') :\vdash a' \centerdot \overline{\mathsf{annot}}(l') \sqsubseteq \overline{\mathsf{annot}}'(l') \\
p_2 &:= \mathsf{monot}_{T'}(p_1) :\vdash T'(a' \centerdot \overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
p_3 &:= \mathsf{justif} :\vdash a \centerdot T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(a' \centerdot \overline{\mathsf{annot}}(l')) \\
p_4 &:= \mathsf{trans}(p_3, p_2) :\vdash a \centerdot T(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
p_5 &:= \mathsf{axiom} :\vdash T(\overline{\mathsf{annot}}(l')) \sqsubseteq T(\overline{\mathsf{annot}}(l')) \\
p_6 &:= \mathsf{axiom} :\vdash a \sqsubseteq a \\
p_7 &:= \mathsf{weak}_{\sqcap}(p_5) :\vdash \textstyle\bigsqcap_{\langle l,l' \rangle \in \mathcal{E}} T_{\langle l,l' \rangle}(\overline{\mathsf{annot}}(l')) \sqsubseteq T(\overline{\mathsf{annot}}(l')) \\
p_8 &:= \mathsf{monot}_{\centerdot}(p_6, p_7) :\vdash a \centerdot \textstyle\bigsqcap_{\langle l,l' \rangle \in \mathcal{E}} T_{\langle l,l' \rangle}(\overline{\mathsf{annot}}(l')) \sqsubseteq a \centerdot T(\overline{\mathsf{annot}}(l')) \\
p_9 &:= \mathsf{trans}(p_8, p_4) :\vdash a \centerdot \textstyle\bigsqcap_{\langle l,l' \rangle \in \mathcal{E}} T_{\langle l,l' \rangle}(\overline{\mathsf{annot}}(l')) \sqsubseteq T'(\overline{\mathsf{annot}}'(l')) \\
\mathsf{goal}(l) &:= \mathsf{intro}_{\sqcap}(\{p_9\}_{\langle l,l' \rangle \in \mathcal{E}}) : \\
&\qquad a \centerdot \textstyle\bigsqcap_{\langle l,l' \rangle \in \mathcal{E}} T_{\langle l,l' \rangle}(\overline{\mathsf{annot}}(l')) \bigsqcap_{\langle l,l' \rangle \in \mathcal{E}} T'_{\langle l,l' \rangle}(\overline{\mathsf{annot}}'(l'))
\end{aligned}$$

Fig. 3.14. Definition of goal for relational certificate translation (case $f = \uparrow$)

Example 3.4.4 *We perform liveness analysis on the variables of the program in Figure 3.13, obtained from the one in Figure 3.12 by node and edge clustering. The intention of the analysis is to provide sufficient information in order to remove assignments to dead variables. The transformed program is given in Figure 3.16. The rest of this subsection is devoted to an explanation of the analysis, and to a justification of the transformation.*

Assuming a standard program semantics, we say that a variable is live at a certain program point if its value will be needed in the future. We refrain from using the classical intensional definition of variable liveness: a variable x *is live at a program node* l *if there is a path from* l *that reaches an expression referring to* x*, without traversing an assignment to* x*. Instead, we prefer to use a more extensional interpretation of liveness, inspired by Benton's Relational Hoare Logic [16], identifying a declaration of a set of live variables as a relational proposition. To this end, we generalize the abstract domain* A

Let $a = \gamma(S(l))$, $a' = \gamma(S(l'))$, $T = T_{\langle l',l\rangle}$ and $T' = T'_{\langle l',l\rangle}$ in:

$$
\begin{aligned}
p_1 &:= \mathsf{goal}(l') : \vdash \overline{\mathsf{annot}}'(l') \sqsubseteq a' \centerdot \overline{\mathsf{annot}}(l') \\
p_2 &:= \mathsf{monot}_{T'}(p_1) : \vdash T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq T'(a' \centerdot \overline{\mathsf{annot}}(l')) \\
p_3 &:= \mathsf{justif} : \vdash T'(a' \centerdot \overline{\mathsf{annot}}(l')) \sqsubseteq a \centerdot T(\overline{\mathsf{annot}}(l')) \\
p_4 &:= \mathsf{trans}(p_2, p_3) : \vdash T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \centerdot T(\overline{\mathsf{annot}}(l')) \\
p_5 &:= \mathsf{axiom} : \vdash T(\overline{\mathsf{annot}}(l')) \sqsubseteq T(\overline{\mathsf{annot}}(l')) \\
p_6 &:= \mathsf{axiom} : \vdash a \sqsubseteq a \\
p_7 &:= \mathsf{weak}_{\sqcup}(p_5) : \vdash T(\overline{\mathsf{annot}}(l')) \sqsubseteq \bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T_{\langle l',l\rangle}(\overline{\mathsf{annot}}(l')) \\
p_8 &:= \mathsf{monot}_{\centerdot}(p_6, p_7) : \vdash a \centerdot T(\overline{\mathsf{annot}}(l')) \sqsubseteq a \centerdot \bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T_{\langle l',l\rangle}(\overline{\mathsf{annot}}(l')) \\
p_9 &:= \mathsf{trans}(p_4, p_8) : \vdash T'(\overline{\mathsf{annot}}'(l')) \sqsubseteq a \centerdot \bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T_{\langle l',l\rangle}(\overline{\mathsf{annot}}(l')) \\
\mathsf{goal}(l) &:= \mathsf{intro}_{\sqcup}(\{p_9\}_{\langle l',l\rangle\in\mathcal{E}}) : \\
&\qquad \bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T'_{\langle l',l\rangle}(\overline{\mathsf{annot}}'(l'))a \centerdot \bigsqcup_{\langle l',l\rangle\in\mathcal{E}} T_{\langle l',l\rangle}(\overline{\mathsf{annot}}(l'))
\end{aligned}
$$

Fig. 3.15. Definition of goal for relational certificate translation (case $f = \downarrow$)

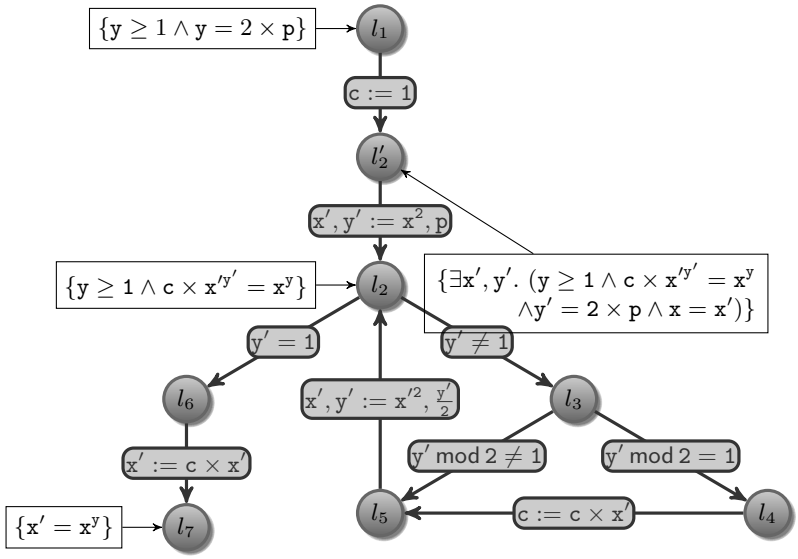


Fig. 3.16. Program after Dead Variable Elimination

of the certificate infrastructure to include relational propositions. An abstract domain A is relational if the associated satisfaction relation $\models_A$ is a subset of $(\mathsf{Env} \times \mathsf{Env}) \times A$. Hence, a relational proposition will be interpreted as a relation on execution environments. Formally, the extension consists in partitioning the domain of variables by attaching to each of them an index $\_\langle 1 \rangle$ or $\_\langle 2 \rangle$. The set of transfer functions is also modified accordingly. For instance, instead of defining the transfer function for the assignment $x{:=}e$ at node l as the substitution $\phi[{}^{e}/_{x}]$, we define it as $\phi[{}^{e\langle 1 \rangle}/_{x\langle 1 \rangle}][{}^{e\langle 2 \rangle}/_{x\langle 2 \rangle}]$, where $e_{\langle i \rangle}$ is the result of indexing every variable occurring at e with $\_\langle i \rangle$.

Then, we define $\gamma(X) = \bigwedge_{v \in X} v_{\langle 1 \rangle} = v_{\langle 2 \rangle}$ as an interpretation of the fact that all variables in X are live. In order to generate a certificate for the optimized program, we apply Proposition 3.21, using as composition operator over relational propositions the function $\centerdot$ defined as

$$\phi \centerdot \psi = \exists x^1, \ldots, x^k.\ \phi[{}^{x^1\langle 2 \rangle}/_{x}] \ldots [{}^{x^k\langle 2 \rangle}/_{x}] \wedge \psi[{}^{x^1\langle 1 \rangle}/_{x}] \ldots [{}^{x^k\langle 1 \rangle}/_{x}]$$

where $\{x^1, \ldots, x^k\}$ are the set of variables in ϕ or ψ. The interpretation of the composition operator is that if X declares the set of live variables, then $\gamma(X) \centerdot \phi$ is the result of existentially quantifying away from ϕ the variables that are not live.

By Proposition 3.14, we know that a certified solution $\langle \gamma \circ \mathsf{live}, \vec{c}'' \rangle$ exists s.t. $\mathsf{live}(l_1) = \{x, y\}$, $\mathsf{live}(l_2') = \{x, y, c\}$ and $\mathsf{live}(l) = \{x, y, c, x', y'\}$ for $l \notin \{l_1, l_2'\}$. Since node l_1 contains an assignment to variables x' and y' and these variables are not live in node l_2', we may safely simplify the statement by removing such assignments. From Proposition 3.21 we can transform the current certified solution by assuming the certificate

$$\mathsf{justif}(l_1, l_2') :\vdash \gamma(\mathsf{live}(l_1)) \centerdot T_{\langle l_1, l_2' \rangle}(\phi) \sqsubseteq T'_{\langle l_1, l_2' \rangle}(\gamma(\mathsf{live}(l_2')) \centerdot \phi)\ .$$

For readability, if ϕ is a non-relational proposition, $\gamma(X) \centerdot \phi$ is equivalently denoted $\exists y_1, \ldots, y_m.\ \phi$ where $\{y_1, \ldots, y_m\} = \mathsf{Var} - X$. Then, the goal of the certificate $\mathsf{justif}(l_1, l_2')$ can be interpreted as $\vdash \phi[{}^{1}/_{c}][{}^{x}/_{x'}][{}^{y}/_{y'}] \sqsubseteq (\exists x', y'.\ \phi)[{}^{1}/_{c}]$.

3.4.5 Concurrency

This section extends the results on certificate transformation obtained for sequential programs in Section 3.3 and Section 3.4 to a concurrent setting. We adopt a verification infrastructure similar to Owicki-Gries logic [56], without proposing criteria to reduce the number of proof obligations. Verification is split in two independent tasks: verifying that a concurrent component satisfies its specification in isolation and verifying that other concurrent components do not invalidate this specification. Since the number of verification conditions is exponential in the size of parallel components, practical applications of Owicki-Gries logic aim to reduce the number of verification conditions. This is done in general by grouping code fragments that are known to be atomically

executed or by omitting proof obligations that are trivially provable. However, we do not consider this issue here.

A concurrent program is defined as the parallel composition of a set of sequential programs, i.e., programs as defined in Definition 3.1.

Definition 3.22 (Concurrent Program). *Let* $\{P_i = \langle \mathcal{N}_i, \mathcal{E}_i, \rightsquigarrow_i, l_{\mathsf{init}i} \rangle\}_{1 \leq i \leq k}$ *be a set of sequential programs. We define the concurrent program* $P_1 \parallel \ldots \parallel P_k$*, i.e., the parallel composition of* $\{P_i\}_{1 \leq i \leq k}$*, as the tuple* $\langle \Pi_{1 \leq i \leq k} \mathcal{N}_i, \mathcal{E}, \rightsquigarrow, l_{\mathsf{init}} \rangle$*, where:*

- $\mathcal{E} = \{\langle (l_1, ..., l_j, ..., l_k), (l_1, ..., l'_j, ..., l_k) \rangle \mid (\forall i \in [1,k].\ l_i \in \mathcal{N}_i) \wedge \langle l_j, l'_j \rangle \in \mathcal{E}_j\}$
- $\langle (l_1, \ldots, l_j, \ldots, l_k), \eta \rangle \rightsquigarrow \langle (l_1, \ldots, l'_j, \ldots, l_k), \eta' \rangle$ *iff* $\langle l_j, \eta \rangle \rightsquigarrow_j \langle l'_j, \eta' \rangle$
- $l_{\mathsf{init}} = (l_{\mathsf{init}1}, \ldots, l_{\mathsf{init}k})$

From the definition, an execution point in the concurrent program $P_1 \parallel \ldots \parallel P_k$ will be determined by the current execution point in each of its components ($\mathcal{N} = \mathcal{N}_1 \times \ldots \times \mathcal{N}_k$). An execution step in any of the program components is considered an execution step of the whole program, i.e., $\langle (l_1, ..., l_j, ..., l_k), (l_1, ..., l'_j, ..., l_k) \rangle \in \mathcal{E}$ for every $\langle l_j, l'_j \rangle \in \mathcal{E}_j$, for any $1 \leq j \leq k$. Finally, the entry point of the composed program is set as the execution point in which each component is at its initial node.

In the sequel, for readability, we consider the parallel composition of two sequential programs P_a and P_b. From the definition, the semantics of a concurrent program $P_\mathsf{a} \parallel P_\mathsf{b}$ is modeled by interleaving the sequential semantics of its components P_a and P_b.

Here we assume that the semantics of P_a and P_b are modeled with the finest level granularity possible in order to capture with the parallel composition every possible semantics interleaving. For instance, if a program is defined as a sequence of atomic instructions, the program representation shall distinguish every interleaving program point as a distinct node. That is, there is exactly one program representation since we cannot merge adjacent nodes.

A standard means to reduce the exponential number of proof obligations is to cluster a set of executable edges into a single one. Clustering nodes l_1 and l_2 into a single node is only possible if l_1 and l_2 are explicitly identified for atomic execution. For instance, when the language features `lock` and `unlock` statements, and nodes l_1 and l_2 are located inside an atomically executable region.

Consider $I_\mathsf{a} = \langle \mathbf{A}, \{T_e\}_{e \in \mathcal{E}_\mathsf{a}}, f \rangle$ and $I_\mathsf{b} = \langle \mathbf{A}, \{T_e\}_{e \in \mathcal{E}_\mathsf{b}}, f \rangle$ the abstract interpretations for the sequential programs P_a and P_b, respectively, where $\mathbf{A}$ is the lattice $\langle A, \sqsubseteq, \sqcap, \sqcup, \top, \bot \rangle$. Let S_a be a labeling for program P_a, such that S_a is a solution of the analysis I_a.

Suppose an execution of P_a in a concurrent environment from the initial label $l_{\mathsf{init}\mathsf{a}}$ to a final label $l_o \in \mathcal{N}_\mathsf{a}$, i.e., $\langle (l_{\mathsf{init}\mathsf{a}}, l), \eta \rangle \rightsquigarrow^\star \langle (l_o, l'), \eta' \rangle$ for some $\eta, \eta' \in \mathsf{Env}$. From the definition of $\rightsquigarrow$ (Def. 3.22), the execution may traverse an arbitrary number of edges in $\mathcal{E}_\mathsf{b}$, affecting the execution of P_a. In that

situation, the soundness of the abstract interpretation I_{a} w.r.t. the semantics of P_{a} does not hold anymore, i.e., we cannot ensure that $\models \eta : S_{\mathsf{a}}(l_{\mathsf{init}_{\mathsf{a}}})$ implies $\models \eta' : S_{\mathsf{a}}(l_o)$.

We extend then the notion of solution of an abstract interpretation to require not only the validity of the labeling with respect to a sequential program execution, but also whether a labeling is stable with respect to the execution of the other components.

For a labeling S_{a}, in addition to be a solution of an abstract interpretation I_{a} (Def. 3.4), namely a local solution of I_{a}, we require S_{a} to be stable w.r.t. (i.e., to be preserved by) the execution of the program P_{b}. We say that a condition a at node $l \in \mathcal{N}_{\mathsf{a}}$ is stable w.r.t. P_{b} and labeling S_{b} if the concurrent execution of P_{b} does not invalidate a as long as P_{b} satisfies S_{b}. More precisely, for every node $l \in \mathcal{N}_{\mathsf{a}}$ and edge $\langle l_{\mathsf{b}}, l'_{\mathsf{b}} \rangle \in \mathcal{E}_{\mathsf{b}}$, we require, assuming the validity of $S_{\mathsf{b}}(l_1)$, that $S_{\mathsf{a}}(l)$ is preserved by the application of the transfer function $T_{\langle l_{\mathsf{b}}, l'_{\mathsf{b}} \rangle}$.

We formalize when a local solution for one component is stable with respect to the behavior of the other components.

Definition 3.23 (globally-stable solution). *A labeling S_{a} for program P_{a}, is a stable solution of I_{a}, w.r.t. program P_{b} with labeling S_{b}, if it is a solution of I_{a} and for every edge $\langle l_{\mathsf{b}}, l'_{\mathsf{b}} \rangle \in \mathcal{E}_{\mathsf{b}}$ and node $l \in \mathcal{N}_{\mathsf{a}}$ the following condition holds:*

- $f = \uparrow$ *and* $S_{\mathsf{b}}(l_{\mathsf{b}}) \sqcap S_{\mathsf{a}}(l) \sqsubseteq T_{\langle l_{\mathsf{b}}, l'_{\mathsf{b}} \rangle}(S_{\mathsf{a}}(l))$ *or,*
- $f = \downarrow$ *and* $T_{\langle l'_{\mathsf{b}}, l_{\mathsf{b}} \rangle}(S_{\mathsf{b}}(l'_{\mathsf{b}}) \sqcap S_{\mathsf{a}}(l)) \sqsubseteq S_{\mathsf{a}}(l)$.

Notice that in contrast to previous section we require labelings S_{a} and S_{b} to be total. To give an intuition of this definition, consider the short code fragment on the left, containing a statement $y{:=}2 \times x$ at node l with successor node l'.

$$\begin{array}{c} \cdots \\ l : \{x \geq 0\} \\ \mathtt{y := 2x} \\ l' : \{y \geq 0\} \\ \cdots \end{array} \qquad \begin{array}{c} \cdots \\ l_{\mathsf{b}} : \{\mathsf{even}(y)\} \\ \mathtt{x = z^y} \\ \cdots \end{array}$$

A labeling S_{a} such that $S_{\mathsf{a}}(l) = x \geq 0$ and $S_{\mathsf{a}}(l') = y \geq 0$, is a solution of a weakest precondition calculus at node l. That is proved by showing the validity of the verification condition $S_{\mathsf{a}}(l) \sqsubseteq \mathsf{wp}_{\langle l, l' \rangle}(S_{\mathsf{a}}(l'))$, that is proving $x \geq 0 \Rightarrow (y \geq 0)[^{2x}/_{y}]$. In a concurrent environment, we must also verify that the conditions $x \geq 0$ and $y \geq 0$ are not invalidated by any other statement. For instance, for the fragment of code of P_{b} shown on the right, node l_{b} contains the statement $x{:=}z^y$, then we must ensure that it does not invalidate $x \geq 0$ or $y \geq 0$, assuming as hypothesis the validity of $S_{\mathsf{b}}(l_{\mathsf{b}})$. Showing that $x \geq 0$ is preserved by the execution of $\mathtt{x = z^y}$, i.e., proving the condition $S_{\mathsf{b}}(l_{\mathsf{b}}) \sqcap S_{\mathsf{a}}(l'_{\mathsf{a}}) \sqsubseteq T_{\langle l_{\mathsf{b}}, l'_{\mathsf{b}} \rangle}(S_{\mathsf{a}}(l'_{\mathsf{a}}))$, is feasible if for instance $S_{\mathsf{b}}(l_{\mathsf{b}})$ is defined as $\mathsf{even}(y)$. The case for the preservation of $y \geq 0$ is straightforward. Similarly,

$S_{\mathsf{b}}(l_2)$ must also be proved stable with respect to the assignment at edge $\langle l, l' \rangle$ of program P_{a}.

We define a labeling for a concurrent program as a tuple of labelings, one for each of the parallel components. Consider the family of sequential programs $\{P_i\}_{1 \leq i \leq k}$ and $\left\{I_i = \langle \mathbf{A}, \{T_e^i\}_{e \in \mathcal{E}_i}, f \rangle\right\}_{1 \leq i \leq k}$ the corresponding family of abstract interpretations. Let the tuple $(S_i)_{0 \leq i \leq k}$ be a labeling for the concurrent program $P_1 \parallel \ldots \parallel P_k$. We define when this labeling is a solution for the tuple of abstract interpretations $(I_i)_{0 \leq i \leq k}$.

Definition 3.24 (Solution for a Concurrent Program). *We say that a labeling $\langle S_1, \ldots, S_k \rangle$ is a solution of the abstract interpretation $(I_i)_{0 \leq i \leq k}$ for the concurrent program $P_1 \parallel \cdots \parallel P_k$, if for every $j \in [1, k]$ the labeling S_j is a solution of I_j (Def. 3.4) and for every $i \in [1, n]$, $i \neq j$, S_j is stable with respect to the program P_i and the labeling S_i.*

Suppose that I_{a} and I_{b} are consistent with the semantics of P_{a} and P_{b} respectively. It follows from Definition 3.24 that, if $(S_{\mathsf{a}}, S_{\mathsf{b}})$ is a solution for $(I_{\mathsf{a}}, I_{\mathsf{b}})$, and that $\langle (l_{\mathsf{inita}}, l_{\mathsf{initb}}), \eta \rangle \rightsquigarrow^{\star} \langle (l_{o_{\mathsf{a}}}, l_{o_{\mathsf{b}}}), \eta' \rangle$ and that $\models \eta : S_{\mathsf{a}}(l_{\mathsf{inita}})$ and $\models \eta : S_{\mathsf{b}}(l_{\mathsf{initb}})$ then $\models \eta' : S_{\mathsf{a}}(l_{o_{\mathsf{a}}})$ and $\models \eta' : S_{\mathsf{b}}(l_{o_{\mathsf{b}}})$.

Motivated by the reasons explained in Section 3.2.3 and assuming the existence of of certificate infrastructures (Def. 3.8), we extend the definition of globally-stable solution with a notion of certificates.

Definition 3.25 (Certified Globally-Stable Solution). *A certified globally-stable solution for I_{a}, w.r.t. program P_{b} with labeling S_{b}, is a triple $\langle S_{\mathsf{a}}, \vec{c}, \vec{c'} \rangle$ where $\langle S_{\mathsf{a}}, \vec{c} \rangle$ is a certified solution of I_{a} (Def. 3.9), and for all $\langle l_{\mathsf{b}}, l'_{\mathsf{b}} \rangle \in \mathcal{E}_{\mathsf{b}}$ and $l \in \mathcal{N}_{\mathsf{a}}$:*

- $f = \uparrow$ *and* $\vec{c'}(l_{\mathsf{b}}, l'_{\mathsf{b}}, l) :\vdash S_{\mathsf{b}}(l_{\mathsf{b}}) \sqcap S_{\mathsf{a}}(l) \sqsubseteq T_{\langle l_{\mathsf{b}}, l'_{\mathsf{b}} \rangle}(S_{\mathsf{a}}(l))$ *or,*
- $f = \downarrow$ *and* $\vec{c'}(l_{\mathsf{b}}, l'_{\mathsf{b}}, l) :\vdash T_{\langle l_{\mathsf{b}}, l'_{\mathsf{b}} \rangle}(S_{\mathsf{b}}(l'_{\mathsf{b}}) \sqcap S_{\mathsf{a}}(l)) \sqsubseteq S_{\mathsf{a}}(l)$.

Certifying analyzers can be extended to a concurrent setting.

Lemma 3.26 (Certifying Analyzers for Globally Stable Solutions). *Consider a solution S_{a} for the abstract interpretation $I_{\mathsf{a}}^{\sharp} = \langle \mathbf{A}^{\sharp}, \{T_e\}_{e \in \mathcal{E}_{\mathsf{a}}}, f \rangle$, where $\mathbf{A}^{\sharp}$ is a lattice $\langle A^{\sharp}, \sqsubseteq^{\sharp}, \sqcap^{\sharp}, \sqcup^{\sharp}, \top^{\sharp}, \bot^{\sharp} \rangle$. Assume for every $a, a' \in A^{\sharp}$ s.t. $a \sqsubseteq^{\sharp} a'$, the certificates $\mathsf{monot}_{\gamma}(a, a')$ and cons defined in Proposition 3.14, and the certificate $\mathsf{distrib}_{(\gamma, \sqcap)}$ defined in Fig. 3.6. Then, one can compute $\vec{c'}$ s.t. $\langle \gamma \circ S, \vec{c}, \vec{c'} \rangle$ is a certified globally stable solution for I_{a} w.r.t. program P_{b} and labeling $\gamma \circ S_{\mathsf{b}} : \mathcal{N}_{\mathsf{b}} \to A$, provided S_{a} is stable w.r.t. P_{b} and S_{b}.*

Proof. In Figure 3.17 we show the definition of $\vec{c'}(l, l', l_1)$ in terms of monot_T and $\mathsf{distr}_{(\gamma, \sqcap)}$ (defined in Figure 3.6), and monot_{γ} and cons (defined in Prop. 3.14). The definition of the certificate $\vec{c}$ follows from Proposition 3.14.

Definition 3.27 (Certified Solution for a Concurrent Program). *Let $(I_i)_{1 \leq i \leq k}$ be certificate infrastructures for the programs $(P_i)_{1 \leq i \leq k}$. A certified*

$f = f^\sharp = \uparrow$:

$$\begin{aligned}
\mathsf{hyp} &:= S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1) \sqsubseteq^\sharp T^\sharp_{\langle l,l'\rangle}(S_\mathsf{a}(l_1))\\
p_1 &:= \mathsf{monot}_\gamma : \vdash \gamma(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1)) \sqsubseteq \gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{a}(l_1)))\\
p_2 &:= \mathsf{distr}_{(\gamma,\sqcap)} : \vdash \gamma(S_\mathsf{b}(l)) \sqcap \gamma(S_\mathsf{a}(l_1)) \sqsubseteq \gamma(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1))\\
p_3 &:= \mathsf{trans}(p_2, p_1) : \vdash \gamma(S_\mathsf{b}(l)) \sqcap \gamma(S_\mathsf{a}(l_1)) \sqsubseteq \gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{a}(l_1)))\\
p_4 &:= \mathsf{cons} : \vdash \gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{a}(l_1))) \sqsubseteq T_{\langle l,l'\rangle}(\gamma \circ S_\mathsf{a}(l_1))\\
\vec{c'}(l,l',l_1) &:= \mathsf{trans}(p_3, p_4) : \vdash \gamma \circ S_\mathsf{b}(l) \sqcap \gamma \circ S_\mathsf{a}(l_1) \sqsubseteq T_{\langle l,l'\rangle}(\gamma \circ S_\mathsf{a}(l_1))
\end{aligned}$$

$f = \downarrow, f^\sharp = \uparrow$

$$\begin{aligned}
\mathsf{hyp} &:= S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1) \sqsubseteq^\sharp T^\sharp_{\langle l,l'\rangle}(S_\mathsf{a}(l_1))\\
p_1 &:= \mathsf{monot}_\gamma : \vdash \gamma(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1)) \sqsubseteq \gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{a}(l_1)))\\
p_2 &:= \mathsf{distr}_{(\gamma,\sqcap)} : \vdash \gamma(S_\mathsf{b}(l)) \sqcap \gamma(S_\mathsf{a}(l_1)) \sqsubseteq \gamma(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1))\\
p_3 &:= \mathsf{trans}(p_2, p_1) : \vdash \gamma(S_\mathsf{b}(l)) \sqcap \gamma(S_\mathsf{a}(l_1)) \sqsubseteq \gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{a}(l_1)))\\
p_4 &:= \mathsf{monot}_T(p_3) : \vdash T_{\langle l,l'\rangle}(\gamma(S_\mathsf{b}(l)) \sqcap \gamma(S_\mathsf{a}(l_1))) \sqsubseteq T_{\langle l,l'\rangle}(\gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{a}(l_1))))\\
p_5 &:= \mathsf{cons} : \vdash T_{\langle l,l'\rangle}(\gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{a}(l_1)))) \sqsubseteq \gamma \circ S_\mathsf{a}(l_1)\\
\vec{c'}(l,l',l_1) &:= \mathsf{trans}(p_4, p_5) : \vdash T_{\langle l,l'\rangle}(\gamma(S_\mathsf{b}(l)) \sqcap \gamma(S_\mathsf{a}(l_1))) \sqsubseteq \gamma \circ S_\mathsf{a}(l_1)
\end{aligned}$$

$f = f^\sharp = \downarrow$

$$\begin{aligned}
\mathsf{hyp} &:= T^\sharp_{\langle l,l'\rangle}(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1)) \sqsubseteq^\sharp S_\mathsf{a}(l_1)\\
p_1 &:= \mathsf{monot}_\gamma : \vdash \gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1))) \sqsubseteq \gamma(S_\mathsf{a}(l_1))\\
p_2 &:= \mathsf{cons} : \vdash T_{\langle l,l'\rangle}(\gamma(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1))) \sqsubseteq \gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1)))\\
p_3 &:= \mathsf{trans}(p_2, p_1) : \vdash T_{\langle l,l'\rangle}(\gamma(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1))) \sqsubseteq \gamma(S_\mathsf{a}(l_1))\\
p_4 &:= \mathsf{distr}_{(\gamma,\sqcap)} : \vdash \gamma(S_\mathsf{b}(l)) \sqcap \gamma(S_\mathsf{a}(l_1)) \sqsubseteq \gamma(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1))\\
p_5 &:= \mathsf{monot}_T(p_4) : \vdash T_{\langle l,l'\rangle}(\gamma(S_\mathsf{b}(l)) \sqcap \gamma(S_\mathsf{a}(l_1))) \sqsubseteq T_{\langle l,l'\rangle}(\gamma(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1)))\\
\vec{c'}(l,l',l_1) &:= \mathsf{trans}(p_5, p_3) : \vdash T_{\langle l,l'\rangle}(\gamma(S_\mathsf{b}(l)) \sqcap \gamma(S_\mathsf{a}(l_1))) \sqsubseteq \gamma(S_\mathsf{a}(l_1))
\end{aligned}$$

$f = \uparrow, f^\sharp = \downarrow$

$$\begin{aligned}
\mathsf{hyp} &:= T^\sharp_{\langle l,l'\rangle}(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1)) \sqsubseteq^\sharp S_\mathsf{a}(l_1)\\
p_1 &:= \mathsf{monot}_\gamma : \vdash \gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1))) \sqsubseteq \gamma(S_\mathsf{a}(l_1))\\
p_2 &:= \mathsf{monot}_T(p_1) : \vdash T_{\langle l,l'\rangle}(\gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1)))) \sqsubseteq T_{\langle l,l'\rangle}(\gamma(S_\mathsf{a}(l_1)))\\
p_3 &:= \mathsf{cons} : \vdash \gamma((S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1))) \sqsubseteq T_{\langle l,l'\rangle}(\gamma(T^\sharp_{\langle l,l'\rangle}(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1))))\\
p_4 &:= \mathsf{distr}_{(\gamma,\sqcap)} : \vdash \gamma \circ S_\mathsf{b}(l) \sqcap \gamma \circ S_\mathsf{a}(l_1) \sqsubseteq \gamma(S_\mathsf{b}(l) \sqcap S_\mathsf{a}(l_1))\\
\vec{c'}(l,l',l_1) &:= \mathsf{trans}(p_4, \mathsf{trans}(p_3, p_2)) : \vdash \gamma \circ S_\mathsf{b}(l) \sqcap \gamma \circ S_\mathsf{a}(l_1) \sqsubseteq T_{\langle l,l'\rangle}(\gamma(S_\mathsf{a}(l_1)))
\end{aligned}$$

Fig. 3.17. Certifying Analyzers for parallel program composition.

solution for the program $P_1 \parallel \ldots \parallel P_k$ *is a tuple* $(\langle S_i, \vec{c}_i, \vec{c'}_i\rangle)_{1\leq i\leq k}$ *such that for each* $j \in [1,k]$, $\langle S_j, \vec{c}_j, \vec{c'}_j\rangle$ *is a certified globally stable solution of* P_j *w.r.t.* P_i *and* S_i *for any other* $i \neq j$.

The existence of certifying analyzers for concurrent programs follows directly from Definition 3.27 and Lemma 3.26.

Transformation of Certificates

In the rest of this section, we extend certificate translation for the transformations characterized in Section 3.4.2, i.e., a program $P'_j = \langle \mathcal{N}'_j, \mathcal{E}'_j, \rightsquigarrow'_j, l_{\mathsf{init}}\rangle$ is a transformation of a program $P_j = \langle \mathcal{N}_j, \mathcal{E}_j, \rightsquigarrow_j, l_{\mathsf{init}}\rangle$ if $\mathcal{N}'_j \subseteq \mathcal{N}_j$ and $\mathcal{E}'_j \subseteq \mathcal{E}_j$. For readability, we consider the case in which only one of the parallel components is transformed, i.e., $P_1 \parallel \ldots \parallel P_j \parallel \ldots \parallel P_k$ is transformed into $P_1 \parallel \ldots \parallel P'_j \parallel \ldots \parallel P_k$. The generalization of the following results to transformations that operate simultaneously (but still independently) in every program component is straightforward.

The following proposition extends certificate transformation (as Proposition 3.17) to a concurrent setting.

Proposition 3.28 (Existence of certificate transformers). *Let* I'_j *be the certificate infrastructure* $I'_j = \langle A, \{T'_e\}_{e\in\mathcal{E}_j}, f\rangle$ *associated to* P'_j. *Assume the existence of the certificates* $\mathsf{assoc}_\sqcap$, $\mathsf{commut}_\sqcap$ *and* $\mathsf{distr}_{(T,\sqcap)}$ *defined in Fig. 3.6. Let* $\langle R, \vec{c}_R, \vec{c'}_R\rangle$ *be a certified globally stable solution of* I_j *s.t. for every* $\langle l, l'\rangle \in \mathcal{E}_j$ *and* $a \in A$

- $\mathsf{justif}(l,l') :\vdash T'_{\langle l,l'\rangle}(a) \sqsubseteq R(l') \sqcap T_{\langle l,l'\rangle}(a)$, *if* $f = \downarrow$; *or*
- $\mathsf{justif}(l,l') :\vdash R(l) \sqcap T_{\langle l,l'\rangle}(a) \sqsubseteq T'_{\langle l,l'\rangle}(a)$ *if* $f = \uparrow$.

Then one can transform every certified solution $(\langle S_i, \vec{c}_i, \vec{c'_i}\rangle)_{1\leq i\leq k}$ *for the analysis* $(I_i)_{1\leq i\leq k}$ *of the program* $P_1 \parallel \ldots \parallel P_j \parallel \ldots \parallel P_k$ *into a certified solution* $(\langle S_1, \vec{d}_1, \vec{d'}_1\rangle, \ldots, \langle S'_j, \vec{d}_j, \vec{d'}_j\rangle, \ldots, \langle S_k, \vec{d}_k, \vec{d'}_k\rangle)$ *for* $(I_1, \ldots, I'_j, \ldots, I_k)$.

Proof. We show that for any $i \neq j$ *one can transform every certified globally stable labelings* $\langle S_i, \vec{c}_i, \vec{c'_i}\rangle$ *for* P_i *and* $\langle S_j, \vec{c}_j, \vec{c'_j}\rangle$ *for* P_j *into the certified labelings* $\langle S_i, \vec{d}_i, \vec{d'_i}\rangle$ *for* P_i *and* $\langle S'_j, \vec{d}_j, \vec{d'_j}\rangle$ *for* P'_j, *where for all* $l \in \mathcal{N}'_j$, $S'_j(l)$ *is defined as* $S_j(l) \sqcap R(l)$.

First of all, notice that for $i \neq j$ *we can let* $\vec{d}_i = \vec{c}_i$ *and that it is not hard to define* $\vec{d'_j}$ *from* $\vec{c'}_j$ *and* $\vec{c'}_R$, *since* $\{T_e\}_{e\in\mathcal{E}_i}$ *have not changed for* $i \neq j$.

Building the certificates $\vec{d}_j$ *that ensures that* S'_j *is a local solution is exactly the same procedure as in Section 3.4. The case for the certificates* $\vec{d'_i}$ *for* $i \neq j$ *can be found in Figures 3.18 and 3.19, for the cases* $f = \uparrow$ *and* $f = \downarrow$, *respectively. Notice that the certification of the solution* $\langle R, \vec{c}_R, \vec{c'}_R\rangle$ *is only required to define the certificates* $\vec{d}_j$ *and* $\vec{d'_j}$.

Let $T = T_{\langle l_j, l'_j \rangle}$ and $T' = T'_{\langle l_j, l'_j \rangle}$ in:

$$
\begin{aligned}
p_1 &:= \vec{c'}_i &&:\vdash S_j(l_j) \sqcap S_i(l) \sqsubseteq T(S_i(l)) \\
p_2 &:= \mathsf{justif}(l_j, l'_j) &&:\vdash R(l_j) \sqcap T(S_i(l)) \sqsubseteq T'(S_i(l)) \\
p_3 &:= \mathsf{weak}_{\sqcap}(p_1) &&:\vdash S_j(l_j) \sqcap S_i(l) \sqcap R(l_j) \sqsubseteq T(S_i(l)) \\
p_4 &:= \mathsf{weak}_{\sqcap}(\mathsf{axiom}) &&:\vdash S_j(l_j) \sqcap S_i(l) \sqcap R(l_j) \sqsubseteq R(l_j) \\
p_5 &:= \mathsf{intro}_{\sqcap}(p_3, p_4) &&:\vdash S_j(l_j) \sqcap S_i(l) \sqcap R(l_j) \sqsubseteq T(S_i(l)) \sqcap R(l_j) \\
\vec{d}_i(l_j, l'_j, l) &:= \mathsf{trans}(p_5, p_2) &&:\vdash S'_j(l_j) \sqcap S_i(l) \sqsubseteq T'(S_i(l))
\end{aligned}
$$

Fig. 3.18. Definition of $\vec{d}_i(l_1, l'_1, l_2)$. Case $f = \uparrow$

Let $T = T_{\langle l_j, l'_j \rangle}$ and $T' = T'_{\langle l_j, l'_j \rangle}$ in:

$$
\begin{aligned}
p_1 &:= \mathsf{axiom} &&:\vdash S_j(l_j) \sqcap S_i(l) \sqsubseteq S_j(l_j) \sqcap S_i(l) \\
p_2 &:= \mathsf{weak}_{\sqcap}(p_1) &&:\vdash S'_j(l_j) \sqcap S_i(l) \sqsubseteq S_j(l_j) \sqcap S_i(l) \\
p_3 &:= \mathsf{monot}_T(p_2) &&:\vdash T(S'_j(l_j) \sqcap S_i(l)) \sqsubseteq T(S_j(l_j) \sqcap S_i(l)) \\
p_4 &:= \vec{c'}_i &&:\vdash T(S_j(l_j) \sqcap S_i(l)) \sqsubseteq S_i(l) \\
p_5 &:= \mathsf{trans}(p_3, p_4) &&:\vdash T(S'_j(l_j) \sqcap S_i(l)) \sqsubseteq S_i(l) \\
p_6 &:= \mathsf{weak}_{\sqcap}(p_5) &&:\vdash R(l'_j) \sqcap T(S'_j(l_j) \sqcap S_i(l)) \sqsubseteq S_i(l) \\
p_7 &:= \mathsf{justif} &&:\vdash T'(S'_j(l_j) \sqcap S_i(l)) \sqsubseteq R(l'_j) \sqcap T(S'_j(l_j) \sqcap S_i(l)) \\
\vec{d'}_i(l_j, l'_j l) &:= \mathsf{trans}(p_7, p_6) &&:\vdash T'(S'_j(l_j) \sqcap S_i(l)) \sqsubseteq S_i(l)
\end{aligned}
$$

Fig. 3.19. Definition of $\vec{d'}_i(l_j, l'_j l)$. Case $f = \downarrow$

3.4.6 Example

Consider the simple imperative syntax to define program components:

$$
\begin{aligned}
c ::= {} & \texttt{skip} \mid x{:=}e \mid c; c \\
& \mid \underline{\texttt{await}}\ b\ \underline{\texttt{then}}\ c \\
& \mid \underline{\texttt{while}}\ b\ \underline{\texttt{do}}\ c\ \underline{\texttt{od}} \\
& \mid \underline{\texttt{if}}\ b\ \underline{\texttt{then}}\ c\ \underline{\texttt{else}}\ c\ \underline{\texttt{fi}} \\
p ::= {} & c \parallel c
\end{aligned}
$$

Statements $x{:=}e$ are executed atomically and statements $\underline{\texttt{await}}\ b\ \underline{\texttt{then}}\ c$ are such that c does not contain `while` nor `await` commands. The command `await` b `then` c suspends its execution until b is satisfied, then it executes c atomically. The statement $\underline{\texttt{wait}}\ b$ is a shorthand for $\underline{\texttt{await}}\ b\ \underline{\texttt{do}}$ `skip`.

Consider as example the verification of the producer-consumer program shown in Figure 3.20. We assume the variables `in` and `out` are initially equal to 0. We represent the producer component as the graph in Figure 3.21.

The labeling S for the graph of Figure 3.21 representing intermediate program annotations is defined as:

Producer :

l_1 : $\underline{\texttt{while}}$ $\texttt{in} < M$ $\{\mathsf{Inv} \wedge \texttt{in} \leq M\}$ $\underline{\texttt{do}}$
l_2 : $\{\mathsf{Inv} \wedge \texttt{in} < M\}$
$\underline{\texttt{wait}}$ $(\texttt{in} - \texttt{out} < N)$
l_3 : $\{\mathsf{Inv} \wedge \texttt{in} < M \wedge \texttt{in} - \texttt{out} < N\}$
$\texttt{buffer}[\texttt{in} \bmod N] := \texttt{a}[\texttt{in}]$
l_4 : $\{\mathsf{Inv} \wedge \texttt{in} < M \wedge \texttt{in} - \texttt{out} < N \wedge \texttt{buffer}[\texttt{in} \bmod N] = \texttt{a}[\texttt{in}]\}$
$\texttt{in} := \texttt{in} + 1$
$\underline{\texttt{od}}$
l_f : $\{\mathsf{Inv} \wedge \texttt{in} = M\}$

Consumer :

$\underline{\texttt{while}}$ $\texttt{out} < M$ $\{\mathsf{Inv}_2 \wedge \texttt{out} \leq M\}$ $\underline{\texttt{do}}$
$\{\mathsf{Inv}_2 \wedge \texttt{out} < M\}$
$\underline{\texttt{wait}}$ $(\texttt{out} < \texttt{in})$
$\{\mathsf{Inv}_2 \wedge \texttt{out} < M \wedge \texttt{out} < \texttt{in}\}$
$\texttt{b}[\texttt{out}] := \texttt{buffer}[\texttt{out} \bmod N]$
$\{\mathsf{Inv}_2 \wedge \texttt{out} < M \wedge \texttt{out} < \texttt{in} \wedge \texttt{b}[\texttt{out}] := \texttt{a}[\texttt{out} \bmod N]\}$
$\texttt{out} := \texttt{out} + 1$
$\underline{\texttt{od}}$
$\{\mathsf{Inv}_2 \wedge \texttt{out} = M\}$

where

$$\mathsf{Inv} \stackrel{\text{def}}{=} \forall k.\ \texttt{out} \leq k < \texttt{in} \Rightarrow \texttt{a}[k] = \texttt{buffer}[k \bmod N]$$
$$\mathsf{Inv}_2 = \mathsf{Inv} \wedge \forall j.\ 0 \leq j < \texttt{out} \Rightarrow \texttt{a}[j] = \texttt{b}[j]$$

Fig. 3.20. Producer-Consumer Program

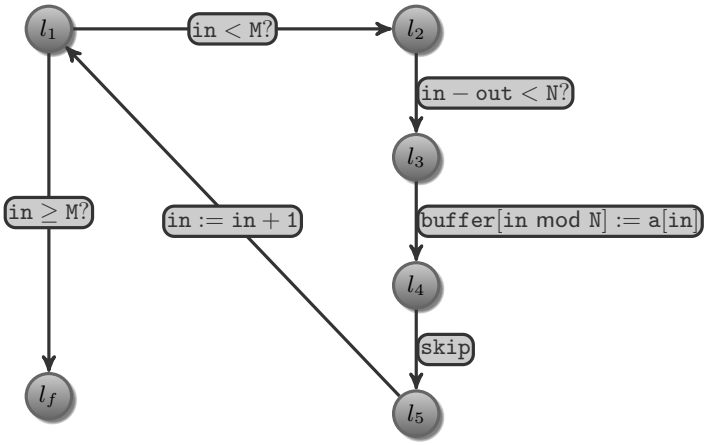


Fig. 3.21. Graph representing the *Producer* component

$$
\begin{aligned}
&S(l_1) = \mathsf{Inv} \wedge \mathtt{in} \leq M\\
&S(l_2) = \mathsf{Inv} \wedge \mathtt{in} < M\\
&S(l_3) = \mathsf{Inv} \wedge \mathtt{in} < M \wedge \mathtt{in} - \mathtt{out} < N\\
&S(l_4) = \mathsf{Inv} \wedge \mathtt{in} < M \wedge \mathtt{in} - \mathtt{out} < N \wedge \mathtt{buffer}[\mathtt{in} \bmod N] = \mathtt{a}[\mathtt{in}]\\
&S(l_5) = S(l_4)\\
&S(l_f) = \mathsf{Inv} \wedge \mathtt{in} = M
\end{aligned}
$$

Among the verification condition to prove the local validity of the annotations, i.e., that S is a solution, we have:

- $S(l_2) \sqsubseteq T_{\langle l_2,l_3\rangle}(S(l_3))$, and
- $S(l_5) \sqsubseteq T_{\langle l_5,l_1\rangle}(S(l_1))$;

that is,

- $\mathsf{Inv} \wedge \mathtt{in} < M \Rightarrow \mathtt{in} - \mathtt{out} < N \Rightarrow \mathsf{Inv} \wedge \mathtt{in} < M \wedge \mathtt{in} - \mathtt{out} < N$, and
- $\mathsf{Inv} \wedge \mathtt{in} < M \wedge \mathtt{in} - \mathtt{out} < N \wedge \mathtt{buffer}[\mathtt{in} \bmod N] = \mathtt{a}[\mathtt{in}]$
  $\Rightarrow \forall k.\ \mathtt{out} \geq k < \mathtt{in} + 1 \Rightarrow \mathtt{a}[k] = \mathtt{buffer}[k \bmod N] \wedge \mathtt{in} + 1 = M$

Among the verification condition to prove the stability of the annotations for the consumer w.r.t. the producer, we have to prove for instance

$$
\begin{aligned}
\mathsf{Inv}_2 \wedge \mathtt{in} - \mathtt{out} < N \wedge \mathtt{buffer}[\mathtt{in} \bmod N] = \mathtt{a}[\mathtt{in}] \sqcap S(l_1)\\
\Rightarrow \mathsf{wp}(\mathtt{out} := \mathtt{out} + 1, S(l_1))
\end{aligned}
$$

that is

$$
\begin{aligned}
\mathsf{Inv}_2 \wedge \mathtt{in} - \mathtt{out} < N \wedge \mathtt{buffer}[\mathtt{in} \bmod N] = \mathtt{a}[\mathtt{in}] \wedge \mathsf{Inv} \wedge \mathtt{in} \geq M\\
\Rightarrow \forall k.\ \mathtt{out} + 1 \geq k < \mathtt{in} \Rightarrow \mathtt{a}[k] = \mathtt{buffer}[k \bmod N]
\end{aligned}
$$

A first simple transformation consists in inserting extra statements that affect a fresh variable $\mathtt{r}$, as shown in Figure 3.22. The motivation of this transformation is to ensure the validity of the condition $\mathtt{r} = \mathtt{in} \bmod N$ enabling, thus, a further transformation. In the graph, the transformation consists in introducing the nodes l, l_6, and l_7, together with the edges $\{\langle l, l_1\rangle, \langle l_5, l_6\rangle, \langle l_5, l_7\rangle\}$. Translating the certificate in this step is straightforward since $\mathtt{r}$ is a fresh variable and hence does not appear in the program annotations. The freshness of $\mathtt{r}$ is formalized by the fact that transfer functions T_e for $e \in \{\langle l, l_1\rangle, \langle l_4, l_5\rangle, \langle l_5, l_6\rangle, \langle l_5, l_7\rangle\}$ are defined as the identity function on the original specification. As a result, the labeling S is extended to l as $S(l) = S(l_1)$ and to l_6, l_7 as $S(l_6) = S(l_7) = S(l_5)$.

We proceed then by explaining in detail the application of *loop induction variable strength reduction* on the expression $\mathtt{in} \bmod N$ with the variable $\mathtt{r}$. We assume an analysis is able to compute a solution R such that $R(l)$ is defined as $\mathtt{r} = \mathtt{in} \bmod N$ for l in $\{l_1, l_2, l_3, l_4\}$. It is not hard to verify that this labeling can be certified, since local verification conditions are valid and is stable with respect to transfer functions on the consumer side (since $\mathtt{r}$ is a fresh variable and $\mathtt{in}$ is local to the producer side).

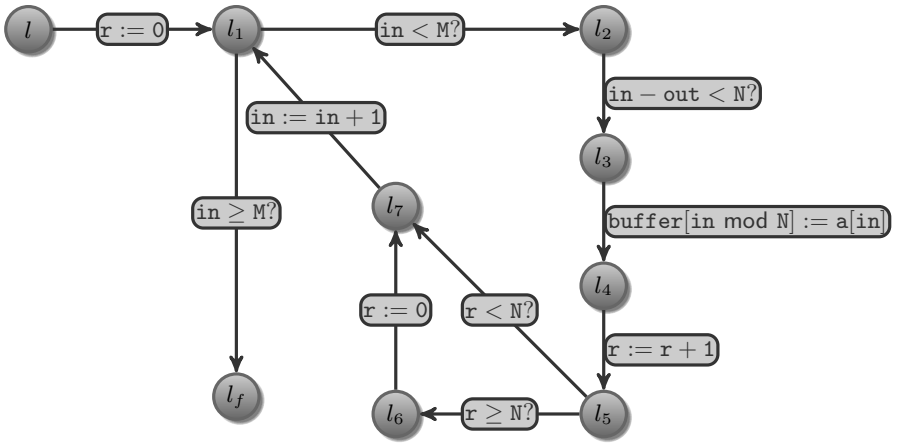


Fig. 3.22. *Producer* component after node insertion

As shown in Figure 3.23, the transformation consists on replacing the assignment $\texttt{buffer}[\texttt{in} \bmod N] := \texttt{a}[\texttt{in}]$ by $\texttt{buffer}[\texttt{r}] := \texttt{a}[\texttt{in}]$.

In a backward certificate infrastructure, by Lemma 3.28, it is sufficient to provide for all $a \in A$ and every edge e a certificate justif s.t.:

$$\mathsf{justif} :\vdash R(l_3) \sqcap T_e(a) \sqsubseteq T'_e(a)$$

In a weakest precondition calculus, since $\langle l_3, l_4 \rangle$ is the only edge that is modified, it is sufficient to provide the certificate for the goal

$$\vdash R(l_3) \wedge \phi\left[{}^{[a|\texttt{in} \bmod N \mapsto \texttt{a}[\texttt{in}]]}/_{a}\right] \sqsubseteq \phi\left[{}^{[a|\texttt{r} \mapsto \texttt{a}[\texttt{in}]]}/_{a}\right]$$

which is valid since $R(l_3) \Rightarrow \texttt{r} = \texttt{in} \bmod N$.

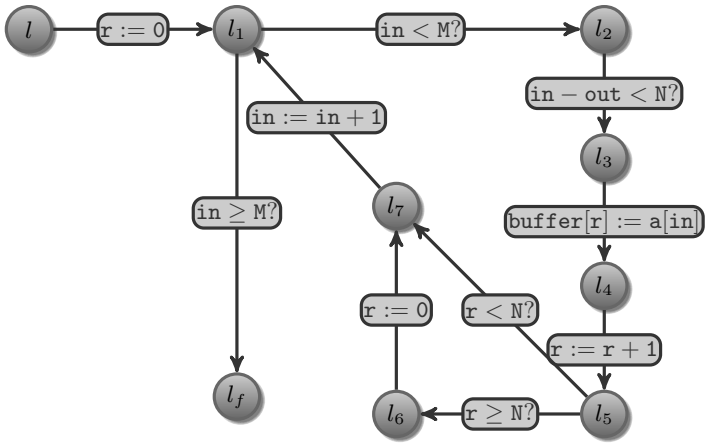


Fig. 3.23. *Producer* component after induction variable strength reduction

Part II

Case Studies

Introduction

Proof compilation as explored in Section 2 targets imperative programs and does not provide support for advanced programming idioms. This part studies the extension of certificate transformation to less typical verification scenarios.

- In a first instance, we consider the case of a simple aspect-oriented paradigm. The main interest of Aspect Oriented Programming (AOP) in a producer-consumer scenario relies on the potential to separate basic functionality from crosscutting concerns. AOP supports an incremental development process, in which the basic functionality is provided by a baseline program, that is successively refined, possible by third parties, with components (namely advices) that deal with non-functional concerns, such as efficiency or security. This incremental development mechanism is an interesting setting for Proof Carrying Code scenarios, since it considers not only the code producer and the code consumer, but untrusted intermediaries that may modify the original code functionality.
  First, a special purpose verification condition generator is proposed for a subclass of advices that preserves the specification of the baseline program. We then define a weaving mechanism that compiles an AOP program to a low-level stack-based language, merging the baseline program with the advices. In the basis of this transformation, we translate the certificates of the baseline program and the certificates of the specification-preserving advices to a certificate for a standard VCgen for the low-level language.
- In a second instance, we consider hybrid verification methods that combine static analyses and verification condition generators. State-of-the-art verification tools are increasingly relying on hybrid approaches to reduce the number and size of verification conditions. In a PCC scenario, this is interesting because it reduces not only the certificate generation effort, but the size of certificates that are sent along with the code. However, since such verification environments operate on source programs, it is necessary to translate the verification results to executable code. In this situation, there are two alternative paths to follow. One can translate the result of the analysis and the certificates for the source program to a hybrid VCgen for executable code. To that end, we show that the result of the analysis is preserved by compilation, and that proof obligations are preserved along non-optimizing compilation, following the ideas of [14, 8]. Alternatively, one can relate hybrid verification to standard verification, transforming the certificates of the hybrid verification environment to a non-hybrid setting. To this end, the result of the analysis is represented as logical formulae and incorporated to the original verification invariants. The first approach is advantageous in the sense that certificates are reduced in size. The second approach can be applied when the client side does not feature a hybrid verification setting, with the cost of increasing the size of certificates. However, the soundness of the hybrid framework

follows from the soundness of the standard VCgen and the existence of a transformation procedure.

- In Chapter 6 we consider a parallel programming language that exploits the capabilities of hierarchical memory architectures. There is an increasing need to provide analysis and verification methods to help developers write, maintain and optimize their high-performance applications. However, verification methods are seldom considered in the context of high-performance computing. Sequoia [30, 37, 35] is a programming language that abandons the traditionally memory model, for a hierarchical, tree-structured, and explicitly managed memory [2, 28, 36]. We first propose, using abstract interpretation [26, 27], a compositional proof system to analyze Sequoia programs and reason about them. The proof system differs from standard frameworks for concurrent programs since it is geared towards the verification of divide-and-conquer applications, in which computations are repeatedly fragmented into smaller tasks that will be executed at lower (and disjoint) levels of the memory hierarchy. Finally, for common program optimizations [37], we show that provably correct Sequoia programs are transformed into provably correct Sequoia programs.

4

Specification Preserving Advice Weaving

4.1 Introduction

In order to study proof compilation for a very simple AOP language, we introduce the notion of specification-preserving advice. Informally, an advice a is specification-preserving for an annotated piece of code $\{\Phi\}c\{\Psi\}$, where Φ and Ψ denote the pre and postcondition for c, respectively, if the advised code $a \triangleright c$ satisfies the same specification, i.e., $\{\Phi\}a \triangleright c\{\Psi\}$. Specification-preserving advices are natural in the context of PCC with intermediaries, since many aspects related to security (resource management, logging, *etc.*) and efficiency (e.g., cached functions, optimized code, *etc.*) fall in this category. Moreover, specification-preserving advices support "separate verification" (as coined by [38]) and allow intermediaries to treat correctness proofs of the baseline code as black-boxes. Concretely, intermediaries will only be required to prove that advices are specification-preserving w.r.t. the code they advise, and an appropriate certificate translator will produce certificates of the weaved code.

We provide a definition of the class of specification-preserving advices that support modular reasoning and a simple static analysis that ensures that advices are specification-preserving. In addition, we propose an algorithm that takes as input an AOP program p and a certificate c of its correctness, and returns a certificate for the compiled program $[\![p]\!]$.

4.2 A basic motivating example

Consider the program p with a procedure main and another procedure twice advised unconditionally by a:

$$\begin{aligned} \mathsf{main}(x) &= y := \mathsf{twice}(x); z := y + x;\ \mathsf{return}\ z \\ \mathsf{twice}(x) &= \mathsf{return}\ (x + x) \\ a(x) &= x := 0; z := \mathsf{proceed}(x);\ \mathsf{return}\ z \end{aligned}$$

Consider a table Γ that associates with each procedure a triple consisting of a precondition, a postcondition, and a modifies clause that states which variables are modified. We choose the obvious specifications for main and twice, i.e.,

$$\Gamma(\mathsf{main}) = (\mathsf{true}, \mathsf{res} = x^\star + x^\star + x^\star, \emptyset)$$
$$\Gamma(\mathsf{twice}) = (\mathsf{true}, \mathsf{res} = x^\star + x^\star, \emptyset)$$

(We consider that the variables x, y and z are local variables, and thus are not declared in the modified clauses).

One can generate for each procedure a verification condition that guarantees that the procedure meets its specification. Both verification conditions hold obviously.

Nevertheless, when the advice code

$$a(x) = x := 0; z := \mathsf{proceed}(x);\ \mathsf{return}\ z$$

is declared to be executed around twice, every call of the function twice starts instead the execution of the advice a. The special statement proceed in the body of a starts the execution of the code under advice, i.e., function twice in this example.

Therefore, when the advice a executes around the function twice, all terminating executions of the program main will simply return the value given as input, and thus the postcondition will not be satisfied if main is called with an input distinct from 0. In this case, the problem is caused by the fact that a forces twice to be executed with input 0. In other words, a is not parameter-preserving, i.e., causes twice to be called with an input different from the one that is declared in the program.

A similar problem shall occur if an advice modifies a global variable that is otherwise unmodified by the procedures it advises. More generally, advices should, in addition to be parameter-preserving, preserve specifications.

Now consider instead the correct advice $a(x)$:

$$(\mathtt{if}\ x \neq 0\ \mathtt{then}\ z := \mathsf{proceed}(x)\ \mathtt{else}\ z := 0);\ \mathsf{return}\ z$$

The function $\hat{a}(x)$ derived from $a(x)$ by replacing the proceed statement by a call to f:

$$(\mathtt{if}\ x \neq 0\ \mathtt{then}\ z := \mathsf{twice}(x)\ \mathtt{else}\ z := 0);\ \mathsf{return}\ z$$

is specification-preserving, since the proof obligation for $\hat{a}$ with with the same pre and postcondition as twice is logically equivalent to

$$x \neq 0 \Rightarrow x + x = x + x \wedge x = 0 \Rightarrow 0 = x + x$$

and it is thus valid. Note that the proof obligations for $\hat{a}$ relies on the specification of twice, but not on its code.

Commands	$c ::= v{:=}e \mid c;\ c \mid v{:=}f(e)$
	$\mid v{:=}$ `proceed`(e)
	$\mid$ `if` b `then` c `else` c
	$\mid$ `while` b `do` c
	$\mid$ `skip` $\mid$ `return` e
Procedures	$proc ::= f\ arg^*\ c_b$
Point-cut descriptors	$ptd ::=$ if b around f
Advices	$advice ::= ptd^+\ a\ arg^*\ c_a$
Programs	$Prog ::= proc^*\ advice^*$

Fig. 4.1. Syntax of SAL programs

4.3 A simple AOP language

This section introduces SAL, a simple procedural language with aspects, i.e., with modules implementing concerns different from the basic functionality. Each aspect is implemented as a collection of code units, namely advices, that are weaved before, after or around the baseline code at syntactic program points as specified by the point-cut descriptors. For simplicity, SAL is restricted to around advices, to point-cuts at procedure calls, and to point-cut descriptors that do not refer to the control-flow graph.

4.3.1 Syntax

The syntax of commands can be found in Figure 4.1, where v ranges over the sets $\mathcal{V}$ of local variables and $\mathcal{X}$ of global variables, arg ranges over local variables, f ranges over the set $\mathcal{F}$ of procedure names, and a ranges over the set $\mathcal{A}$ of advice names. A baseline command is a command that does not contain any proceed command. We let c_b and c_a range respectively over baseline and advice commands.

Point-cut descriptors are of the form if b around f, where b is a boolean condition and f is a procedure name. Then, each procedure is composed of an identifier, its formal parameters and a command that represents its body. Each advice is composed of an identifier from a set $\mathcal{A}$ of advice names, a non-empty set of point-cut descriptors, its formal parameters, and an extended command that represents its body. A program is a given by a set of procedures with a distinguished main procedure and a set of advices.

4.3.2 Semantics

Advice weaving, which enables aspects to influence the execution of programs at designated program points and under certain conditions, is the fundamental mechanism that determines the semantics of AOP programs. Thus, the essence

of SAL programs is captured by the transition rules for the commands call and proceed, which are described informally below.

Upon reaching a call statement of the form $v{:=}f(e)$, one checks in the order prescribed by the declaration of advices whether the guard of a point-cut descriptor for f is satisfied. If there is no point-cut descriptor for f such that the guard is satisfied, then one executes f; otherwise, if a is the first advice for f whose guard is satisfied, then one executes the body of a.

Upon reaching a statement of the form $v{:=}\ \texttt{proceed}(e)$, one must examine the call stack to determine the current procedure, say f, and the current advice, say a. Then one checks for all advices that occur after a in the declaration of advices whether the guard of a point-cut descriptor for f is satisfied. If there is no point-cut descriptor for f such that the guard is satisfied, then one executes the body of f; otherwise, if a' is the first advice for f whose guard is satisfied, then one executes the body of a'.

Notice that under such a semantics, the body of f will not be executed whenever a procedure call to f triggers an advice that does not contain any proceed statement.

Formally, the semantics of advice weaving is defined by compilation to an intermediate language SBL, defined in Section 4.6. For the purpose of the next sections, it is sufficient to know that the semantics of SAL programs can be modeled by judgments of the form $p, \mu \Downarrow v, \nu$ which read: the execution of program p with initial memory μ terminates with final memory ν and returns value v.

$$\frac{}{\mathsf{vcg}(\texttt{skip}, \varphi) = (\varphi, \emptyset)} \qquad \frac{}{\mathsf{vcg}(x{:=}e, \varphi) = (\varphi[{}^{e}\!/_{x}], \emptyset)}$$

$$\frac{(\varphi_2, S_2) = \mathsf{vcg}(c_2, \varphi) \qquad (\varphi_1, S_1) = \mathsf{vcg}(c_1, \varphi_2)}{\mathsf{vcg}(c_1;c_2, \varphi) = (\varphi_1, S_1 \cup S_2)}$$

$$\frac{}{\mathsf{vcg}(\texttt{return}\ e, \varphi) = (\varphi[{}^{e}\!/_{\mathsf{res}}], \emptyset)}$$

$$\frac{(\varphi_1, S_1) = \mathsf{vcg}(c_1, \varphi) \qquad (\varphi_2, S_2) = \mathsf{vcg}(c_2, \varphi)}{\mathsf{vcg}(\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \varphi) = (b \Rightarrow \varphi_1 \wedge \neg b \Rightarrow \varphi_2, S_1 \cup S_2)}$$

$$\frac{(\varphi', S) = \mathsf{vcg}(c, Inv)}{\mathsf{vcg}(\texttt{while}\ b\ \{Inv\}\ \texttt{do}\ c, \varphi) = (Inv, \{Inv \Rightarrow (b \Rightarrow \varphi' \wedge \neg b \Rightarrow \varphi)\} \cup S)}$$

$$\frac{\Gamma(f) = (\Phi, \Psi, \mathcal{W})}{\mathsf{vcg}(x{:=}f(e), \varphi) = \Phi[{}^{e}\!/_{\mathsf{in}_f}] \wedge (\forall_{\mathcal{W}', \mathsf{res}}.\Psi[{}^{e}\!/_{\mathsf{in}_f}][{}^{\mathcal{W}'}\!/_{\mathcal{W}}][{}^{\mathcal{W}}\!/_{\mathcal{W}^*}] \Rightarrow \varphi[{}^{\mathsf{res}}\!/_{x}][{}^{\mathcal{W}'}\!/_{\mathcal{W}}], \emptyset)}$$

$$\frac{\mathsf{vcg}(x{:=}f(e), \varphi) = (\phi', S)}{\mathsf{vcg}_f(x{:=}\ \texttt{proceed}(e), \varphi) = (\phi', S)}$$

Fig. 4.2. Weakest Precondition Function

4.4 Verification of baseline code

In this section, we briefly introduce a verification method for baseline programs. Each procedure is specified in terms of a pre and postcondition and a frame condition that specifies which variables are modified during the execution of f.

Preconditions are propositions that refer to the functions' formal parameters and to the global variables. Loop invariants and postconditions refer to the initial and current state of global variables (respectively with starred and standard variables). Postconditions may also refer to the function result (with the special variable res).

Each precondition Φ yields a predicate over states, denoted $\mu \models \Phi$ for a state μ, whereas a postcondition Ψ yields a ternary relation over an initial state, a final state, and a result, denoted $\mu, \nu, v \models \Psi$ for the states μ and ν and the value v. Likewise, loop invariants yield binary relations over an initial and a current state.

In order to reason effectively about programs, we assume that all `while` loops carry an invariant (we use $\texttt{while}_I(b)\{s\}$ to denote the while loop annotated with invariant I), and that we dispose of a specification table Γ that associates to each procedure f a triple $(\Phi, \Psi, \mathcal{W})$ where Φ is a precondition, Ψ is a postcondition, and $\mathcal{W}$ is a *modifies* clause that declares all variables that are modified during the execution of f. Furthermore, we let $\mathcal{V}_\Gamma$ be the set of variables that appear in the specification of baseline procedures.

From specification table Γ and an annotated procedure f, one can compute a set $\mathsf{PO}_\Gamma(f)$ of verification conditions, using an extended predicate transformer vcg. Formally, the set $\mathsf{PO}_\Gamma(f)$ is defined as $\Delta_f \cup \{\Phi \Rightarrow \Phi'[{}^{y}/_{y^\star}]\}$, where $\varphi[{}^{e}/_{x}]$ stands for the substitution of the expression e for the free occurrences of variable x in the logic formula φ, $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$, y stands for every variable in $\mathcal{V}_\Gamma$ and $(\Phi', \Delta_f) = \mathsf{vcg}(c, \Psi)$, where c is the body of f. The formal definition of vcg is given in Figure 4.2.

For the verification method to be sound, we must also check the correctness of the *modifies* clause. We assume a sound but incomplete automatic analysis that checks its correctness.

The weakest precondition calculus is sound in the sense that if a program p is valid w.r.t. a specification table Γ with a main procedure specified by (Φ, Ψ), then all executions of p initiated with a memory μ satisfying Φ will terminate with a final memory ν and value v such that (μ, ν, v) satisfy Ψ.

Lemma 4.1 (Soundness). *Let p be a baseline program over a set $\mathcal{F}$ of procedures. Let Γ be a specification table for p and let $\Gamma(\mathsf{main}) = (\Phi, \Psi, \mathcal{W})$. Assume that p is valid w.r.t. Γ. Then, if $p, \mu \Downarrow v, \nu$ and $\mu \models \Phi$, then $\mu, \nu, v \models \Psi$.*

In the setting of PCC, we require that proof obligations come equipped with independently checkable certificates of their validity. For simplicity, we rely on an abstract notion of certificate. Let p be an annotated baseline program and Γ be a specification table. Then, a certificate for the program p w.r.t. Γ

is an indexed set of certificates $(c_\delta)_{\delta \in \mathsf{PO}_\Gamma(f), f \in \mathcal{F}}$ such that $c_\delta :\vdash \delta$ for all δ belonging to $\mathsf{PO}_\Gamma(f)$ and for all procedures f. If such a certificate exists, we say that p is certified w.r.t. Γ.

If a program p is certified w.r.t. a specification table Γ, then it is obviously valid w.r.t. Γ.

4.5 Verifying AOP programs

The purpose of this section is to define a method to verify specification-preserving advices.

Throughout this section, we consider a program p in which all procedures are annotated, i.e., have loop invariants, and specified in a table Γ.

4.5.1 Specification-preserving advices

We extend the verification condition generator to proceed statements, interpreting the proceed statement as a call to the advised function; see Figure 4.2.

Definition 4.2. *An advice a with guard b preserves the specification of method f w.r.t. Γ if it satisfies the specification $(b \wedge \Phi, \Psi, \mathcal{W}')$ where $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$, and $\mathcal{W}' \cap \mathcal{V}_\Gamma \subseteq \mathcal{W}$.*

The condition $\mathcal{W}' \cap \mathcal{V}_\Gamma \subseteq \mathcal{W}$ states that the advice a only modifies variables in $\mathcal{W}$, unless they do not appear on the specification of the baseline program. Let $\mathsf{PO}_{\Gamma,f}(a)$ stand for the proof obligations required to prove that a is specification-preserving w.r.t. f and Γ. If $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$ and c is the body of a, the set $\mathsf{PO}_{\Gamma,f}$ is defined as $\Delta_{a,f} \cup \{\Phi \Rightarrow \phi[y/y^\star]\}$ where $(\phi, \delta_{a,f}) = \mathsf{vcg}(c, \Psi)$ and $y^\star$ stands for every starred variable in ϕ.

We say that an advice a is valid if for all procedures f that it advises, the set of proof obligations $\mathsf{PO}_{\Gamma,f}(a)$ is valid. Then, we say that the program p is valid if all its procedures and all its advices are valid.

We now state soundness of the verification method in the presence of advice weaving.

Lemma 4.3 (Soundness). *Let (p, Γ) be a valid annotated program. Then, if $p, \mu \Downarrow v, \nu$ and $\mu \models \Phi$, then $\mu, \nu, v \models \Psi$.*

We require that certified programs come equipped with a certificate that advices are specification-preserving.

Remark. We can extend the language under consideration with a richer set of point-cut descriptors, for instance to point-cut descriptors that refer to the control-flow graph. To this end, as an alternative to reasoning about the control-flow graph or the call-stack in our logic, we propose a stronger definition of specification preserving advices. An advice a is specification-preserving w.r.t. f and Γ if it satisfies the specification $(\Phi, \Psi, \mathcal{W}')$ where $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$, and $\mathcal{W}' \cap \mathcal{V}_\Gamma \subseteq \mathcal{W}$. Notice that, in contrast to previous definition, the guard b does not appear in the precondition of a.

4.5.2 Example

In this section, we consider an extended program syntax. Let the procedure $g \doteq \texttt{slowRetrieve}$ of a SAL program p be such that, when given as parameter the integer *address* `i`, returns the value `mem`[`i`], where `mem` is a global array variable representing a slow access memory.

Consider the auxiliary global array variables `available` and `cache` and the SAL procedures $f_1 \doteq \texttt{updateCache}$ and $f_2 \doteq \texttt{isAvailable}$. Let ϕ stand for the consistency of the `cache` variable with respect to the array `availability`, i.e., $\phi \doteq \forall i.(\texttt{available}[i] \Rightarrow \texttt{cache}[i] = \texttt{mem}[i])$. For simplicity, assume that the variables `available` and `cache` are only accessible by these procedures.

Assume that $\Gamma(g) = (\Phi, \Psi, \mathcal{W})$ where $\Phi \doteq 0 \leq \texttt{i} < N \wedge \phi$, $\Psi \doteq \mathsf{res} = \texttt{mem}[\texttt{i}] \wedge \phi$ and $\mathcal{W} = \emptyset$.

Procedures f_1 and f_2 are specified with their respective pre and postconditions:

$$\begin{aligned}
&\Phi_1 \doteq \Phi \\
&\Psi_1 \doteq \texttt{cache} = \texttt{cache}^{\star}[\texttt{i} \mapsto \texttt{v}] \wedge \phi \\
&\Phi_2 \doteq 0 \leq \texttt{i} < N \\
&\Psi_2 \doteq \mathsf{res} = \texttt{available}[i]
\end{aligned}$$

Introducing the advice $a \doteq \texttt{fastRetrieve}$ improves the store access time by taking advantage of the array variables `available` and `cache` and the procedures f_1 and f_2. This advice receives as parameter the address `i` and returns the *cached* value if available or, otherwise, allows the function g to proceed:

```
around slowRetrieve(Address i) fastRetrieve {
    b:= isAvailable(i);
    if b
        return cache[i]
    else
        v:=proceed(i);
        updateCache(i, v);
        return v
}
```

One can prove that a is specification preserving by showing that the proposition

$$\begin{aligned}
&\Phi_2 \wedge \forall_b.(\Psi_2[{}^{b}\!/_{\mathsf{res}}] \Rightarrow \\
&\quad (b \Rightarrow \Psi[{}^{\texttt{cache[i]}}\!/_{\mathsf{res}}] \wedge \phi) \wedge \\
&\quad (\neg b \Rightarrow \Phi \wedge \forall_{\mathsf{res}}.(\Psi \Rightarrow \\
&\qquad (\Phi_1 \wedge \forall_{\texttt{cache}'}.(\Psi_1[{}^{\texttt{cache}'}\!/_{\texttt{cache}}][{}^{\texttt{cache}}\!/_{\texttt{cache}^{\star}}] \Rightarrow (\Psi \wedge \phi)[{}^{\texttt{cache}'}\!/_{\texttt{cache}}])))))
\end{aligned}$$

is implied by Φ.

4.5.3 Harmless advices

In general, it is not decidable whether an advice a preserves the specification of a procedure f w.r.t. a specification table Γ. Therefore, it is of interest to develop automated approximate methods to detect specification-preserving advices. A natural condition is to require that the advice does not modify the variables in $\mathcal{V}_\Gamma$ and always executes a proceed statement. Since such requirements are closely related to the notion of Harmless Advice [29], we call such advices specification-harmless.

The set of SAL commands is extended with assertions $\texttt{assert}(\phi)$ and ghost assignments $\texttt{set}\ z' := z$, where ϕ is a proposition and z' is a ghost variable not appearing in the original program. The definition of vcg is extended accordingly:

$$\begin{aligned}\mathsf{vcg}(\texttt{assert}(\phi), \varphi) &= (\phi, \{\phi \Rightarrow \varphi\})\\ \mathsf{vcg}(\texttt{set}\ z' := e, \varphi) &= (\phi[{}^{e}/_{z'}], \emptyset)\end{aligned}$$

Formally, an advice a with parameters $\vec{y}$ and guard b is specification-harmless w.r.t. f and Γ if the procedure $\hat{a}$ whose body is obtained from the body of a by substituting $x{:=}\ \texttt{proceed}(\vec{e})$ by

$$\texttt{assert}(\vec{z^\star} = \vec{z}); x{:=}f(\vec{e}); \texttt{set}\ x', \vec{z'} := x, \vec{z}$$

satisfies the specification

$$(b \wedge \Phi, x' = \mathsf{res} \wedge \vec{z'} = \vec{z}, \mathcal{W}')$$

where $\Gamma(f) = (\Phi, \Psi, \mathcal{W})$, and $\mathcal{W}' \cap \mathcal{V}_\Gamma = \emptyset$, and where $x', \vec{z'}$ are fresh ghost variables, and where $\vec{z}$ is an enumeration of $\mathcal{V}_\Gamma$. We classify an advice as *control flow preserving* if every path in its control flow contains exactly one proceed statement. We assume the existence of an automated approximate static analysis to check this condition.

Lemma 4.4. *Assume a is control-flow preserving advice. Then, if a is specification harmless with respect to f and Γ, then it is specification-preserving.*

Dantas and Walker [29] propose a mechanism to check that the execution of an advice does not interfere with the final value produced by the computation of the baseline procedure. It consists on a type-effect system inspired on information flow type systems that does not consider timing nor termination behavior. One can use this type system as a static analysis to detect whether an advice is specification-harmless.

4.5.4 Beyond harmless advices

There are many natural examples of advices that do not necessarily trigger a proceed statement. For example, advices that seek to improve efficiency by replacing a procedure call by a semantically equivalent but more efficient

computation will not call a proceed statement. For such examples of advices, it is still possible to use the property of specification-harmless to ensure that the advice is specification-preserving for those paths in which a proceed statement is effectively called, and generate a proof obligation for all paths that do not call to proceed.

Recall the advice of the basic example shown in Section 4.2:

$$a(x) = (\texttt{if } x \neq 0 \texttt{ then } z := \texttt{proceed}(x) \texttt{ else } z := 0);\ \texttt{return } z$$

Clearly, we have two possible execution paths depending on whether the input value is equal to 0. To verify that a preserves the specification of f, i.e., $(\mathsf{true}, \mathsf{res} = x^\star + x^\star)$, we consider each possible path separately. In case that the parameter x is not equal to 0 we know that exactly one `proceed` statement will be executed, that no variable is modified and that the expression returned by the proceed statement is passed unchanged by the advice. Thus, we can use a simple static analysis to detect whether this path is specification-harmless. However, the path corresponding to an input equal to 0 does not execute a `proceed` statement, so we need to generate proof obligations that ensures that the specification is still preserved. In this case, it corresponds to the valid proposition $x = 0 \Rightarrow 0 = x + x$.

4.6 Compiling advices

From an applicative perspective, AOP is transparent and compilers target typical back-ends: indeed, it is the role of the compiler to integrate these concerns into a single executable object, through a weaving mechanism that modifies the code of each procedure depending on the advices that operate over it. In this section, we define the compilation of SAL programs to a stack-based language.

4.6.1 Target language

The target language is a simple stack-based language (SBL). The syntax of SBL instructions is given in Figure 4.3, where v and l ranges over integers, x ranges over program variables, *cmp* over relations between integer values, and g ranges over function names. A SBL program consists of a set of function names, and for each function g a declaration of the form $g\ args^* = instr^*$. The operational semantics of SBL programs is standard, and defined by a small-step relation $\rightsquigarrow$ between states. A state is either final, in which case it consists of a global memory μ and a result value v, or intermediary, in which case it consists of a global memory μ and a list of frames lf, each frame consisting of the name of the function being called, of a program counter, of a local memory with a distinguished variable *par* that stores the parameter of

$$
\begin{array}{rl}
instr ::= & \mathsf{nop} \mid \mathsf{push}\ v \mid \mathsf{load}\ x \mid \mathsf{store}\ x \\
& \mid \mathsf{jmp}\ l \mid \mathsf{jmpif}\ cmp\ l \\
& \mid \mathsf{invoke} \mid \mathsf{return}
\end{array}
$$

Fig. 4.3. Instruction set for SBL

the function being called, and of an operand stack. Figure 4.4 gives the rules for invoke and return instructions, where $[par \mapsto v]$ denotes the local memory that only assigns v to par.

$$
\frac{p_f[i] = \mathsf{invoke}\ f}{\langle \mu, \langle f', pc, lm, v : os\rangle :: lf\rangle \rightsquigarrow \langle \mu, \langle f, 1, [par \mapsto v], \epsilon\rangle :: \langle f', pc+1, lm, os\rangle :: lf\rangle}
$$

$$
\frac{p_f[i] = \mathsf{return}}{\langle \mu, \langle f, pc, lm, v : os\rangle :: \langle f', pc', lm', os'\rangle :: lf\rangle \rightsquigarrow \langle \mu, \langle f', pc', lm', v :: os'\rangle :: lf\rangle}
$$

Fig. 4.4. Operational semantics of SBL

4.6.2 Compiler

The compiler for SAL programs is defined in Figure 4.5 as a function $[\![\,]\!]$ that takes a command and returns a list of labeled instructions. The compiler function $[\![\,]\!]_{\mathsf{e}}$ takes an integer expression e and returns a sequence of instructions whose effect is to push on top of the stack the evaluation of the expression e. The compiler function $[\![\,]\!]_{\mathsf{b}}$ takes, in addition to a boolean expression b, a label l and outputs a sequence a instructions that forces the program execution to jump to the program point labeled l if the condition b evaluates to true. The compiler for commands is standard, to the exception of the function call and the `proceed` statement, whose compilation involves advice weaving. Since SBL does not feature a dedicated mechanism for advice weaving, each advice is compiled multiple times, exactly once per procedure it advises, and the procedure call $x{:=}f(e)$ is compiled into

$$[\![e]\!]_{\mathsf{e}} :: \mathsf{invoke}\ \hat{a}_f :: \mathsf{store}\ x$$

where a is the first advice for f, and $\hat{a}_f$ is its specific compilation for f. The code of $\hat{a}_f$ is of the form

$$[\![b, l]\!]_{\mathsf{b}} :: \mathsf{load}\ par :: \mathsf{invoke}\ \hat{a}'_f :: \mathsf{return} :: [l : a_f]$$

where a_f is obtained by compilation from a by translating any proceed statement of the form $x{:=}\ \texttt{proceed}(e)$ by

$$[\![e]\!] :: \mathsf{invoke}\ a'_f :: \mathsf{store}\ x$$

where a' is the next advice for f. In other words, the code of $\hat{a}_f$ tests if the guard for a holds, and if so proceeds to execute the body of the advice, or lets $\hat{a}'_f$ proceed otherwise.

$$
\begin{array}{l}
[\![\texttt{skip}]\!] = [l : \mathsf{nop}] \\
[\![x{:=}e]\!] = \mathsf{let}\ \mathsf{ins}_e{=}[\![e]\!]_{\mathsf{e}}\ \mathsf{in} \\
\qquad\qquad \mathsf{ins}_e :: \mathsf{store}\ x \\
[\![c_1;c_2]\!] = \mathsf{let}\ \mathsf{ins}_1{=}[\![c_1]\!]\ \mathsf{in} \\
\qquad\qquad \mathsf{let}\ \mathsf{ins}_2{=}[\![c_2]\!]\ \mathsf{in} \\
\qquad\qquad \mathsf{ins}_1 :: \mathsf{ins}_2 \\
[\![\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2]\!] = \\
\quad \mathsf{let}\ \mathsf{ins}_1{=}[\![c_1]\!]\ \mathsf{in} \\
\quad \mathsf{let}\ \mathsf{ins}_2{=}[\![c_2]\!]\ \mathsf{in} \\
\quad \mathsf{let}\ \mathsf{ins}_b{=}[\![b, l_1]\!]_{\mathsf{b}}\ \mathsf{in} \\
\quad \mathsf{ins}_b :: \mathsf{ins}_2 :: \mathsf{jmp}\ l :: [l_1 : \mathsf{ins}_1] :: [l : \mathsf{nop}] \\
[\![\texttt{while}\ b\ \texttt{do}\ c]\!] = \\
\ \ \mathsf{let}\ \mathsf{ins}_c{=}[\![c]\!]\ \mathsf{in} \\
\ \ \mathsf{let}\ \mathsf{ins}_b{=}[\![b, l_c]\!]_{\mathsf{b}}\ \mathsf{in} \\
\ \ \mathsf{jmp}\ l :: [l_c : \mathsf{ins}_c] :: [l : \mathsf{ins}_b] \\
[\![x{:=}h(e)]\!]^f = \mathsf{let}\ \mathsf{ins}_e{=}[\![e]\!]_{\mathsf{e}}\ \mathsf{in} \\
\qquad\qquad\qquad \mathsf{ins}_e :: \mathsf{invoke}\ a_f :: \mathsf{store}\ x \\
[\![\texttt{return}\ e]\!] = \mathsf{let}\ \mathsf{ins}{=}[\![e]\!]_{\mathsf{e}}\ \mathsf{in} \\
\qquad\qquad\qquad \mathsf{ins} :: \mathsf{return} \\
[\![x{:=}\ \texttt{proceed}(e)]\!]^a_f = \mathsf{let}\ \mathsf{ins}_e{=}[\![e]\!]_{\mathsf{e}}\ \mathsf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathsf{ins}_e :: \mathsf{invoke}\ a'_f :: \mathsf{store}\ x
\end{array}
$$

Fig. 4.5. Compiler for SAL programs

In order to achieve the desired effect, the compiler is thus parametrized by a procedure (used in the clause for procedure calls to trigger the appropriate advice), or by a procedure and an advice (used in the clause for proceed to trigger the appropriate advice). For readability, we use superscripts to indicate the parameter and omit the superscript in all cases where it is not used.

4.7 Certificate translation

In this section, we show that a valid SAL program is compiled into a valid SBL program. To this end, we first define a verification method for SBL programs. The method is strongly inspired from earlier work, and in particular [14].

4.7.1 Verification of SBL programs

stack expressions $\bar{os} ::= \mathbf{s} \mid \bar{e} :: \bar{os} \mid \uparrow^k \bar{os}$
logical expressions $\bar{e} ::= \mathsf{res} \mid x^\star \mid x \mid c \mid \bar{e}\ op\ \bar{e} \mid \bar{os}[k]$

Fig. 4.6. Logical SBL expressions

The extended set of logical expressions is defined in Figure 4.6. In the definition, $\mathbf{s}$ is a special variable representing the current operand stack and $\uparrow^k \bar{os}$ denotes the stack $\bar{os}$ minus its k-first elements. An annotation is a proposition that does not contain stack sub-expressions. An annotated bytecode instruction is either a bytecode instruction or a proposition and a bytecode instruction: $\bar{i} ::= i \mid (\phi, i)$ An annotated program is a pair (p, Γ), where p is a bytecode program in which some instructions are annotated and Γ is a specification table that associates to each procedure f a triple $(\Phi, \Psi, \mathcal{W})$ where Φ is a precondition, Ψ is a postcondition, and $\mathcal{W}$ is a *modifies* clause that declares all variables that may be modified during the execution of f.

Verification of SBL programs is defined in terms of a weakest precondition function wp that operates on annotated programs. In order for the wp function to be well-defined, we must restrict our attention to well-annotated programs [7, 14, 57], i.e., programs in which all cycles in the control-flow graph must pass through an annotated instruction. We characterize such programs by an inductive definition.

An annotated program p is well-annotated if every procedure is well annotated. A formal definition of well annotated programs can be found in Definition 2.1.

Given a well-annotated procedure, one generates an assertion for each label, using the assertions that were given or previously computed for its successors. This assertion represents the precondition that an initial state should satisfy for the procedure to terminate only in a state satisfying its postcondition.

Let (p, Γ) be a well-annotated program.

- The weakest precondition calculus over (p, Γ) is defined in Figure 4.7.
- The set $\mathsf{PO}(f)$ of verification conditions of the procedure f is sucht that:

$$\frac{}{\Phi \Rightarrow \mathsf{wp}_{\mathcal{L}}(0)[\vec{x^\star}/\vec{x}] \in \mathsf{PO}_\Gamma(f)} \qquad \frac{f[k] = (\phi, i)}{\phi \Rightarrow \mathsf{wp}_i(k) \in \mathsf{PO}_\Gamma(f)}$$

An annotated SBL program is valid w.r.t. Γ if for very procedure f the proof obligations $\mathsf{PO}_\Gamma(f)$ are valid.

let $\Gamma(f)=(\Phi,\Psi,\mathcal{W})$ and y represent every variable in $\mathcal{W}$:

$$
\begin{array}{lcll}
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{c::\mathbf{s}}/_{\mathbf{s}}] & \text{if } g[k]=\mathsf{push}\ c\\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{(\mathbf{s}[0]\ op\ \mathbf{s}[1])::\uparrow^2\mathbf{s}}/_{\mathbf{s}}] & \text{if } g[k]=\mathsf{binop}\ op\\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{x::\mathbf{s}}/_{\mathbf{s}}] & \text{if } g[k]=\mathsf{load}\ x\\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)[^{\uparrow\mathbf{s},\mathbf{s}[0]}/_{\mathbf{s},x}] & \text{if } g[k]=\mathsf{store}\ x\\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(l) & \text{if } g[k]=\mathsf{jmp}\ l\\
\mathsf{wp}_i(k) & = & (\mathbf{s}[0]\neq 0\Rightarrow \mathsf{wp}_{\mathcal{L}}(k+1)[^{\uparrow^1\mathbf{s}}/_{\mathbf{s}}])\wedge & \text{if } g[k]=\mathsf{jmpif}\ l\\
 & & \quad \mathbf{s}[0]=0\Rightarrow \mathsf{wp}_{\mathcal{L}}(l)[^{\uparrow^1\mathbf{s}}/_{\mathbf{s}}]) & \\
\mathsf{wp}_i(k) & = & \Psi[^{\mathbf{s}[0]}/_{\mathsf{res}}] & \text{if } g[k]=\mathsf{return}\\
\mathsf{wp}_i(k) & = & \Phi[^{\mathbf{s}[0]}/_{\mathtt{in}}]\wedge(\forall \mathsf{res},y'.\Psi[^{\mathbf{s}[0]}/_{\mathtt{in}}][^{y}/_{y^\star}][^{y'}/_{y}] & \text{if } g[k]=\mathsf{invoke}\ f\\
 & & \quad \Rightarrow \mathsf{wp}_{\mathcal{L}}(k+1)[^{\mathsf{res}::\mathbf{s}}/_{\mathbf{s}}][^{y'}/_{y}]) & \\
 & & & \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \phi & \text{if } g[k]=\phi:i\\
\mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{otherwise}
\end{array}
$$

Fig. 4.7. Weakest precondition for SBL programs

4.7.2 Preservation of validity

The purpose of this section is to prove that valid SAL programs are compiled into valid SBL programs. To this end, we first extend the compiler of Section 4.6 so that compiled programs are well-annotated. This is achieved by modifying the compiler clause for loops:

$$
\begin{array}{rl}
\llbracket \mathtt{while}_I(b)\{c\}\rrbracket = & \text{let } \mathsf{ins}_c=\llbracket c\rrbracket \text{ and } \mathsf{ins}_b=\llbracket b,l_c\rrbracket \text{ in}\\
 & \mathsf{jmp}\ l :: [l_c:\mathsf{ins}_c] :: [l:(I,\mathsf{ins}_b)]
\end{array}
$$

where we denote (I,ins_b) the sequence of instructions obtained by annotating the first instruction of ins_b with I. In the rest of this section, for any SBL function g, we denote $g[l,l']$ the sequence of instructions $g[l] :: g[l+1] :: \ldots :: g[l'-1]$.

Lemma 4.5. *Assuming the axioms* $(v :: \mathbf{s})[0] = v$ *and* $\uparrow (v :: \mathbf{s}) = \mathbf{s}$ *for stacks, the auxiliary compilers* $\llbracket\rrbracket_{\mathsf{e}}$ *and* $\llbracket.\rrbracket_{\mathsf{b}}$ *satisfy the following properties:*

i) for every integer expression e *and function* g *such that* $g[l,l'] = \llbracket e\rrbracket_{\mathsf{e}}$, $\mathsf{wp}_{\mathcal{L}}(l)$ *is equivalent to* $\mathsf{wp}_{\mathcal{L}}(l')[^{e::\mathbf{s}}/_{\mathbf{s}}]$;

ii) for every boolean expression b *and function* f *such that* $g[l,l''] = \llbracket b,l'\rrbracket_{\mathsf{b}}$, $\mathsf{wp}_{\mathcal{L}}(l)$ *is equivalent to*

$$b \Rightarrow \mathsf{wp}_{\mathcal{L}}(l') \wedge \neg b \Rightarrow \mathsf{wp}_{\mathcal{L}}(l'')$$

Given a specification table Γ for SAL programs, Γ' is a specification table for SBL programs extending Γ if for every advice a and procedure f advised by a, $\Gamma'(\hat{a}_f) = (\Phi_f,\Psi_f,\mathcal{W}_f)$ and $\Gamma'(a_f) = (\Phi_f\wedge b,\Psi_f,\mathcal{W}_f)$, where

$\Gamma(f) = (\Phi_f, \Psi_f, \mathcal{W}_f)$. In the following paragraphs, we implicitly consider the specification tables Γ and Γ' respectively for the verification of SAL and SBL programs.

Lemma 4.6. *Let g be an SBL function such that $g[l, l'] = [\![c]\!]$, and let $(\phi, S) = \mathsf{vcg}(c, \mathsf{wp}_{\mathcal{L}}(l'))$. Then, $\phi' \equiv \mathsf{wp}_{\mathcal{L}}(l)$ and the proof obligations in S are equivalent to the proof obligations corresponding to the annotated instructions in $g[l, l']$.*

Consider an SBL program p' compiled from an annotated SAL program p. The following result states that if p is a valid SAL program w.r.t. Γ, then p' is a valid SBL program w.r.t. Γ'.

Theorem 4.7. *Suppose that (p, Γ) is a valid annotated program. That is, for every procedure f and for every advice a, the sets of proof obligations Δ_f and $\mathsf{PO}_{\Gamma,f}(a)$ are valid. Then, for every function f, a_f, and $\hat{a}_f$, the sets $\mathsf{PO}_{\Gamma'}(f)$, $\mathsf{PO}_{\Gamma'}(a_f)$ and $\mathsf{PO}_{\Gamma'}(\hat{a}_f)$ contain valid proof obligations.*

Furthermore, we can prove that a SAL programs certified with respect to Γ is compiled into a SBL program certified with respect to Γ'. More precisely, using the rules of the proof algebra extended with the axioms $(v :: \mathbf{s})[0] = v$ and $\uparrow(v :: \mathbf{s}) = \mathbf{s}$, for every equivalent proof obligations δ and δ', we can transform a certificate c_δ for δ to a certificate $c_{\delta'}$ for δ'. Therefore, if for every procedure $f \in \mathcal{F}$, $(c_\delta)_{\delta \in \mathsf{PO}_\Gamma(f)}$ and $(c_\delta)_{\delta \in \mathsf{PO}_{\Gamma,f}(a)}$ are indexed sets of certificates for a SAL program p, then for every function g of p' we can generate a certificate for the proof obligation $\delta \in \mathsf{PO}_{\Gamma'}(g)$.

4.8 Increasing the Power of Verification

Consider the following trivial example:

$$a_1(x) = z := \mathsf{proceed}(x + 1);\ \mathsf{return}\ z$$
$$a_2(x) = z := \mathsf{proceed}(x - 1);\ \mathsf{return}\ z$$

When executed in isolation around a function f, it is clear that neither a_1 nor a_2 preserves the behavior of f. However, when both are executed around f they collaborate, and the effect of a_1 is neutralized by the effect of a_2.

Then, since it may seem a bit restrictive to require that every advice in its own is specification-preserving, we propose a more general proof system to study instead whether a sequence of advices is specification preserving.

When verifying the behavior of a sequence of advices $\vec{a}$ executing around a function f, we are interested in verifying a specification for the sequence $\vec{a}$ around f (denoted $\vec{a} \triangleright f$), in addition to verifying each advice in isolation. As with functions and advices, the specification for sequences of advices executing around a function f consists of a precondition, a postcondition, and a set of modifiable variables. This specification is inferred and proved from the

specification of its components. For notational convenience, $\vec{a}$ may also stand for an empty sequence of advices.

For each nonempty sequence of advices $\vec{a_1}a\vec{a_2}$ executing around a function f, we call the sequence $\vec{a_2} \triangleright f$, i.e., the advices remaining to be executed around f when a executes a `proceed` statement, an *execution context* of a.

Verification proceeds in two steps. First, each advice a is verified in isolation, i.e., without considering the set of contexts in which the advice a may be executed. To this end, we must rely on a single specification for the expected behavior of the execution invoked by a `proceed` statement. In a second phase, for each context in which the advice may be executed, we check the consistency of the specification for the proceed statement w.r.t. the specification derived for the remaining context.

Verification of advices in isolation.

We extend the specification of advices such that for every advice a we have, in addition to the tuple $(\Phi, \Psi, \mathcal{W})$, a specification for the code that may be invoked by a `proceed` statement. That enables to reason about the correctness of an advice abstracting from the possible contexts in which this advice may be invoked. The specification extension for an advice a consists on an extra and distinct tuple $(\Phi', \Psi', \mathcal{W}')$, in addition to the tuple $(\Phi, \Psi, \mathcal{W})$. The tuple $(\Phi', \Psi', \mathcal{W}')$ is such that $\mathcal{W}'$ specifies the set of variables that the code invoked by a proceed statement is allowed to modify, and Φ' and Ψ' are the pre and postconditions of such invocation, respectively. The propositions Φ' and Ψ' may refer, in addition to the input and output arguments of a (`in` and `res`), to the input and output arguments of the invoked code, represented with the new variables `in′` and `res′`, respectively. It is the goal of the second phase to check, for every context in which the advice a may be executed, that the code allowed to proceed satisfies the specification $(\Phi', \Psi', \mathcal{W}')$.

The predicate transformer `wp` is extended to deal with `proceed` statements, s.t. $\mathtt{wp}_a(x := \mathtt{proceed}(e), \phi)$ is defined as

$$({\Phi'}_a[{}^{e}/_{\mathtt{in}'_a}] \wedge \forall_{y', \mathtt{res}'} . {\Psi'}_a[{}^{e}/_{\mathtt{in}'_a}][{}^{y'}/_{y}][{}^{y}/_{y^{\star\prime}}] \Rightarrow \phi[{}^{res'}/_{x}][{}^{y'}/_{y}][{}^{e}/_{\mathtt{in}'_a}], S)$$

where $(\Phi', \Psi', \mathcal{W}')$ correspond to the specification extension for the `proceed` statement and $y \in \mathcal{W}'$.

By using this modified `wp` function we can prove that the body of an advice satisfies its specification as long as the code invoked by a `proceed` statement satisfies the specification $(\Phi', \Psi', \mathcal{W}')$.

Verifying weaved code.

After statically determining the sequence of advices $\vec{a}_f$ executing around f, we are interested in identifying a set of sufficient proof obligations that ensures that the sequence $\vec{a}_f$ is specification-preserving.

The collection of proof obligations is defined by induction on the length of the sequence of advices $\vec{a}_f$ executing around the procedure f. Since we do

not require that every subsequence $\vec{a_f}'$ of advices preserves the specification, we generalize and accept the inference of pre and postconditions Φ and Ψ for $\vec{a_f}' \triangleright f$ without requiring Φ and Ψ to be compatible with the pre and postcondition of f. The goal of the verification for each subsequence $\vec{a}$ of $\vec{a}_f$ is a judgment of the form $\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\}\vec{a} \triangleright f\{\Psi\}$. For such a judgment, we do not require Φ and Ψ to be compatible with the pre and postcondition of f, i.e., the subsequence $\vec{a}$ is not necessarily specification-preserving.

To verify a judgment $\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\}\vec{a} \triangleright f\{\Psi\}$, we proceed by induction on the length of the sequence $\vec{a}$ to identify the set of proof obligations $\Delta_{\vec{a}}(\Phi, \Psi)$.

In the base case, i.e., when no advice is executed around the function f, we have the judgment $\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\}f\{\Psi\}$ without premises, where Φ and Ψ are the pre and postconditions of f.

Given a non-trivial sequence $\vec{a} = a\vec{a}'$, we consider two alternative sets of verification conditions, depending on whether we can statically ensure that the code of the advice a is *control flow preserving*. We assume an automated static mechanism to check this condition.

In case that it cannot be checked whether a is control-flow preserving we apply the following rule:

$$\frac{\begin{array}{c}\Gamma_{\mathsf{a}}(a) = \langle(\Phi_a, \Psi_a, \mathcal{W}_a), (\Phi'_a, \Psi'_a, \mathcal{W}'_a)\rangle \\ \Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi'\}\vec{a}' \triangleright f\{\Psi'\} \\ \Phi'_a \Rightarrow \Phi'[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}] \qquad \Psi'[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{\mathsf{res}'}/_{\mathsf{res}}] \Rightarrow \Psi'_a \qquad \mathcal{W}_f \cup \mathcal{W}_{\vec{a}'} \subseteq \mathcal{W}'_a\end{array}}{\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi_a\}a\vec{a}' \triangleright f\{\Psi_a\}}$$

For simplicity, we are not considering the boolean condition specified in the point-cut descriptor.

Unfortunately, the rule above makes hard to propagate the information carried by the specification (Φ', Ψ'), unless it is explicitly stated in the specification (Φ_a, Ψ_a) of a. However, under the hypothesis that a is a *control flow preserving* advice we can apply the following alternative rule:

$$\frac{\begin{array}{c}\Gamma_{\mathsf{a}}(a) = \langle(\Phi_a, \Psi_a, \mathcal{W}_a), (\Phi'_a, \Psi'_a, \mathcal{W}'_a)\rangle \\ \Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi'\}\vec{a}' \triangleright f\{\Psi'\} \\ \Phi \Rightarrow \Phi_a \wedge \forall x'.(\Phi'_a[{}^{x'}/_{x}] \Rightarrow \Phi'[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{x'}/_{x}]) \qquad \mathcal{W}_f \cup \mathcal{W}_{\vec{a}'} \subseteq \mathcal{W}'_a \\ \Psi'[{}^{\mathtt{in}'_a}/_{\mathtt{in}_\theta}][{}^{\mathsf{res}'}/_{\mathsf{res}}][{}^{y^\star{}'}/_{y^\star}] \Rightarrow \Psi'_a \wedge \forall x'.(\Psi_a[{}^{\mathtt{in}}/_{\mathtt{in}_a}][{}^{x'}/_{x}] \Rightarrow \Psi[{}^{x'}/_{x}])\end{array}}{\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\}a\vec{a}' \triangleright f\{\Psi\}}$$

where x' represents the global variables potentially modified by a, and W'_a specifies the variables that may be modified by the execution triggered by the `proceed` statement.

For every procedure f advised by $\vec{a}_f$, we define $\Delta_{\vec{a}_f}(\Phi, \Psi)$ as the set of proof obligations required to derive the judgment $\Gamma, \Gamma_{\mathsf{a}} \vdash \{\Phi\}\vec{a}_f \triangleright f\{\Psi\}$. Assume the specification table Γ is such that $\Gamma(f) = (\Phi_f, \Psi_f, \mathcal{W})$. Then, we say that the sequence $\vec{a}_f$ is specification preserving with respect to f, Γ and Γ_{a}, if $\Phi_f \Rightarrow \Phi$, $\Psi \Rightarrow \Psi_f$ and the proof obligations in $\Delta_{\vec{a}_f}(\Phi, \Psi)$ are valid.

Lemma 4.8. *Let p be a SAL program over a set $\mathcal{F}$ of procedures and a set $\mathcal{A}$ of advices. Let Γ be a specification table for $\mathcal{F}$ and Γ_a be a specification table for $\mathcal{A}$. Assume that for every procedure f that is advised by $\vec{a}_f$, the sequence $\vec{a}_f$ is specification preserving with respect to f, Γ and Γ_a. Then, if $f, \mu \Downarrow v, \nu$ and $\mu \models \Phi$, then $\mu, \nu, v \models \Psi$, where Φ and Ψ are the pre and postconditions of f.*

The dynamic nature of some point-cut descriptors can make static verification a difficult task. Consider for example a cflow point-cut descriptor, for which program semantics must refer to a collecting call stack to decide whether a cflow condition is valid.

Although possible, it is cumbersome to reason explicitly about the call stack in the program logic. We propose, thus, the following simple derivation rule to reason in the presence of cflow point-cut descriptors:

$$\frac{\Gamma, \Gamma_\mathsf{a} \vdash \{\Phi\} a \vec{a}' \triangleright f \{\Psi\} \qquad \Gamma, \Gamma_\mathsf{a} \vdash \{\Phi\} \vec{a}' \triangleright f \{\Psi\}}{\Gamma, \Gamma_\mathsf{a} \vdash \{\Phi\} a \overset{\mathsf{cflow}}{\triangleright} (\vec{a}' \triangleright f) \{\Psi\}}$$

where $a \overset{\mathsf{cflow}}{\triangleright} (\vec{a}' \triangleright f)$ denotes that the execution of the advice a is conditional on a cflow statement. The rule can be interpreted as the fact that the specification (Φ, Ψ) is still verifiable with respect to the sequence $a \overset{\mathsf{cflow}}{\triangleright} (\vec{a}' \triangleright f)$, regardless of whether the cflow condition is valid. Although incomplete, this rule may prove to be useful as long as the advice a is specification preserving with respect to (Φ, Ψ).

We have formally proved the soundness of the proof system proposed in this section. In addition, we have shown how to extended the compiler with a mechanism to translate a certificate of correctness of a SAL program to a certificate for the compiled code.

5

Preservation of Proof Obligations in Hybrid Verification Environments

In this section, we consider a small imperative language with arrays, and we focus on a hybrid method based on a generic numerical analysis, inspired by [47, 19], and that can be instantiated to several numeric domains, including polyhedra.

The VCgen exploits the information of the analysis in two useful ways: on the one hand, verification conditions that originate from spurious edges in the control-flow graph are discarded: more precisely, the VCgen ignores the case of out-of-bound accesses whenever the analysis ensures that accesses are within bounds. This leads to fewer, smaller verification conditions. Furthermore, the VCgen adds the results of the analysis as additional assumptions to help the user prove the verification conditions. This is particularly useful for the relational analyses considered as they can provide part of the invariants required to prove programs correct.

Then, we prove preservation of proof obligations using the techniques of [14, 8]. The proof relies on knowing that the solutions of the analysis are preserved by compilation. Although analyzing compiled programs is known to be less precise than analyzing source programs, see e.g. [45], we achieve preservation of solutions by defining at bytecode level an analysis that performs a symbolic execution of stacks, as in [70, 69, 19].

Finally, to relate hybrid verification to standard verification, we show that programs that are provably correct using our hybrid method, remain provably correct using standard verification condition generation. To this end, the compiler translates a hybrid specification (combing logical assertions and analysis results) into a logical one, by giving a logical interpretation of the analysis results.

5.1 Setting

This section introduces the source language (an imperative language with arrays of integers), the target language (a stack-based language with jumps), and the compiler.

We assume two disjoint sets V_s of scalar variables and V_a of array variables, and let V denote $V_s + V_a$. Each variable in V_a has an associated size. Furthermore, we assume two sets V_s^{old} and V_a^{old} in 1-1 correspondence with V_s and V_a, which are used to store initial values. We also consider a special variable res, which is used to represent the value of the program result. Finally, we assume a set $\mathbf{Lab} \subset \mathbb{N}$ of labels.

Source Language

Programs are defined as commands, and are decorated with labels in order to express analysis results:

$$\begin{aligned} e &::= e\ \texttt{op}\ e \mid n \mid x \mid a[e] \\ c &::= \texttt{Skip} \mid [x{:=}e]^k \mid [a[e]{:=}e]^k \mid c; c \mid [\texttt{return}\ e]^k \\ &\quad \mid \texttt{if}\ [e \bowtie e]^k\ \texttt{then}\ c\ \texttt{else}\ c \\ &\quad \mid \texttt{while}\ [e \bowtie e]^k\ \texttt{do}\ c \end{aligned}$$

where x, a, n, and k range over V_s, V_a, $\mathbb{Z}$, and **Lab**, respectively, op ranges over (binary) arithmetic operations, and $\bowtie$ over arithmetic comparisons. We assume that labels occur at most once in commands.

The semantics of source programs is formalized by a small-step transition relation between states. States may be intermediate, in which case they consist of a statement and of a memory, or final, in which case they consist of a memory, and possibly a tag to denote abnormal termination. Memories are modeled as pairs of mappings from variables to values and from arrays to indices to values, respectively. We assume that each array a comes equipped with its size $|a|$ and define the semantic domains of the source language as follows:

$$\begin{aligned} VMem &= V_s \to \mathbb{Z} \\ AMem &= \Pi a \in V_a.\ \{i \mid 1 \le i \le |a|\} \to \mathbb{Z} \\ Mem &= VMem \times AMem \\ State_I^s &= Stmt \times VMem \times AMem \\ State_F^s &= VMem \times AMem \times (\mathbb{Z} + \{\mathbf{AOB}\}) \\ State^s &= State_I^s + State_F^s \end{aligned}$$

The operational semantics of programs is standard and, thus, omitted. (See the next subsection for the semantics of instructions that manipulate arrays).

Bytecode Language

A bytecode program is defined as a list of instructions. Instructions either manipulate the memory that stores the values of variables and the contents of

arrays, or manipulate the operand stack, or perform a conditional or unconditional jump. The set of instructions is defined by the following grammar:

$$\begin{aligned} ins ::= {} & \texttt{prim op} \mid \texttt{push } v \mid \texttt{load } x \mid \texttt{store } x \mid \texttt{return} \\ & \mid \texttt{aload } a \mid \texttt{astore } a \mid \texttt{cjmp} \bowtie l \mid \texttt{jmp } l \mid \texttt{nop} \end{aligned}$$

We denote by $\dot{p}[l]$ the instruction at position l of a bytecode program $\dot{p}$. The semantics of bytecode programs is formalized using a transition relation between states. States may either be intermediate or final; intermediate states consist of a program counter, an operand stack that stores the results of intermediate computations, and a memory. The semantic domains of the bytecode language are defined as follows, where we implicitly assume that the program counter is within the bounds of programs.

$$\begin{aligned} Stack &= \mathbb{Z}^\star \\ State^b_I &= \mathbb{N} \times VMem \times AMem \times Stack \\ State^b_F &= VMem \times AMem \times (\mathbb{Z} + \{\mathbf{AOB}\}) \\ State^b &= State^b_I + State^b_F \end{aligned}$$

The operational semantics of programs is standard. We only provide the operational semantics of the instructions `aload` and `astore`; these instructions may cause abrupt termination if array accesses are out-of-bound. The rules are given in Figure 5.1, where we use the notation $[f \mid r \to s]$ to refer the function that is identical to f everywhere except in r that returns s, for any sets R and S and any function $f : R \to S$.

$$\frac{P[i] = \texttt{aload } a \qquad 0 \leq n < |a|}{\langle i, \rho_\mathsf{v}, \rho_\mathsf{a}, n :: s\rangle \rightsquigarrow \langle i+1, \rho_\mathsf{v}, \rho_\mathsf{a}, \rho_\mathsf{a}\ a\ n :: s\rangle}$$

$$\frac{P[i] = \texttt{aload } a \qquad \neg\ 0 \leq n < |a|}{\langle i, \rho_\mathsf{v}, \rho_\mathsf{a}, n :: s\rangle \rightsquigarrow_{EX} \langle \rho, v, \rho_\mathsf{a}, \mathsf{AOB}\rangle}$$

$$\frac{P[i] = \texttt{astore } a \qquad 0 \leq n < |a|}{\langle i, \rho_\mathsf{v}, \rho_\mathsf{a}, n :: v :: s\rangle \rightsquigarrow \langle i+1, \rho_\mathsf{v}, [\rho_\mathsf{a} \mid a \to [\rho_\mathsf{a}\ a \mid n \to v]], s\rangle}$$

$$\frac{P[i] = \texttt{astore } a \qquad \neg\ 0 \leq n < |a|}{\langle i, \rho_\mathsf{v}, \rho_\mathsf{a}, n :: v :: s\rangle \rightsquigarrow_{EX} \langle \rho_\mathsf{v}, \rho_\mathsf{a}, \mathsf{AOB}\rangle}$$

$$\frac{P[i] = \mathsf{return}}{\langle i, \rho_\mathsf{v}, \rho_\mathsf{a}, n : s\rangle \rightsquigarrow \langle \rho_\mathsf{v}, \rho_\mathsf{a}, n\rangle}$$

Fig. 5.1. Semantics of bytecode (excerpts)

Compiler

The compiler is standard, and defined in Figure 5.2; we use the function $init : \mathbf{Stm} \to \mathbf{Lab}$ to associate to each statement its initial label. We assume *label compatibility*, i.e., that the label of a source statement is the same as the label of the program point for its compilation.

Throughout the rest of the chapter, we let P be a source program, and the bytecode program $\dot{p}$ the result of the compilation of program P.

$$
\begin{array}{l}
[\![n]\!]_{\mathrm{e}} = \mathtt{push}\ n \\
[\![x]\!]_{\mathrm{e}} = \mathtt{load}\ x \\
[\![x[e]]\!]_{\mathrm{e}} = [\![e]\!]_{\mathrm{e}};\ \mathtt{aload}\ x \\
[\![e_1\ \mathtt{op}\ e_2]\!]_{\mathrm{e}} = [\![e_2]\!]_{\mathrm{e}};\ [\![e_1]\!]_{\mathrm{e}};\ \mathtt{prim}\ \mathtt{op} \\
\\
[\![[x{:=}e]^k]\!] = k : [\![e]\!]_{\mathrm{e}}; \mathtt{store}\ x \\
[\![[a[e_1]{:=}e_2]^k]\!] = k : [\![e_2]\!]_{\mathrm{e}};\ [\![e_1]\!]_{\mathrm{e}};\ \mathtt{astore}\ a \\
[\![s1; s2]\!] = [\![s_1]\!];\ [\![s_2]\!] \\
[\![[\mathtt{return}\ e]^k]\!] = k : [\![e]\!]_{\mathrm{e}};\ \mathtt{return} \\
[\![[\mathtt{Skip}]^k]\!] = k : \mathtt{nop} \\
[\![\mathtt{if}\ [e_1 \bowtie e_2]^k\ \mathtt{then}\ s_1\ \mathtt{else}\ s_2]\!] = \\
\quad k : [\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}};\ \mathtt{cjmp} \bowtie k_1; k_2 : [\![s_2]\!]; \mathtt{jmp}\ l; k_1 : [\![s_1]\!] \\
\qquad \text{where } k_1 = init(s_1) = k_2 + |[\![s_2]\!]| + 1 \\
\qquad\qquad k_2 = init(s_2) = k + |[\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}}| + 1 \\
\qquad\qquad l = k_1 + |[\![s_1]\!]| \\
[\![\mathtt{while}\ [e_1 \bowtie e_2]^k\ \mathtt{do}\ s]\!] = \\
\quad k : [\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}}; \mathtt{cjmp} \bowtie k_1; \mathtt{jmp}\ l; k_1 : [\![s]\!]; \mathtt{jmp}\ k \\
\qquad \text{where } k_1 = k + |[\![e_2]\!]_{\mathrm{e}}| + |[\![e_1]\!]_{\mathrm{e}}| + 2 \\
\qquad\qquad l = k_1 + |[\![s]\!]| + 1
\end{array}
$$

Fig. 5.2. Compiler

5.2 Preservation of solutions

It is folklore that compilation potentially yields a loss of precision for relational analyses. The purpose of this section is to show that solutions of abstract interpretations are preserved by compilation, provided one uses symbolic expressions, as done in [70, 69, 19], to mitigate the presence of the operand stack and to recover the loss of precision incurred by compilation.

Symbolic Expressions

Expressions and guards serve as the interface with the numerical relational domain in the analysis for bytecode. Below we let x range over V.

$$\begin{aligned} \mathit{Expr} \ni e &::= n \mid x \mid x[e] \mid ? \mid ?[e] \mid e \ \mathsf{op}\ e \\ \mathit{Guard} \ni t &::= e \bowtie e \end{aligned}$$

The expression ? represents an unknown value; therefore, expressions are interpreted as sets of possible values. Formally, the semantics $[\![e]\!]_\rho$ and $[\![t]\!]_\rho$ of expressions with respect to an environment $\rho = \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}} \rangle$ are defined by the clauses:

$$\begin{aligned} [\![n]\!]_\rho = \{n\} \quad [\![x]\!]_\rho = \rho_{\mathsf{v}}\ x \quad [\![?]\!]_\rho = \mathbb{Z} \quad [\![?[e]]\!]_\rho &= \mathbb{Z} \\ [\![x[e]]\!]_\rho &= \{\rho_{\mathsf{a}}\ x\ v \mid v \in [\![e]\!]_\rho\} \\ [\![e_1\ \mathbf{op}\ e_2]\!]_\rho &= \{n_1\ \mathbf{op}\ n_2 \mid n1 \in [\![e_1]\!]_\rho,\ n_2 \in [\![e_2]\!]_\rho\} \\ [\![e_1 \bowtie e_2]\!]_\rho &\Longleftrightarrow \exists\ n_1 \in [\![e_1]\!]_\rho,\ n_2 \in [\![e_2]\!]_\rho.\ n_1 \bowtie n_2 \end{aligned}$$

As one can see later from the symbolic execution of stack expressions, the expression ? is not required for analyzing bytecode programs that are obtained by compilation of source programs. That is because the stack representation is empty after storing a value in an array.

Abstract domain

Following Miné [47], we assume an abstract numerical domain interface, which can be instantiated with standard relational abstract domains. The interface consists of a domain $\mathbb{D}$ equipped with a partial order $\sqsubseteq \subseteq \mathbb{D} \times \mathbb{D}$, meet and join operators $\sqcap, \sqcup : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$, a least element $\bot$ and a greatest element $\top$. We also assume abstract assignment functions $[\![x{:=}e]\!]^\sharp, [\![x[e_1]{:=}e_2]\!]^\sharp : \mathbb{D} \to \mathbb{D}$, and a function $\mathit{assume}^\sharp$ that maps guards to abstract elements.

Finally, we assume a monotone concretization function $\gamma : \mathbb{D} \to \mathcal{P}(\mathit{VMem} \times \mathit{AMem})$ mapping abstract elements to sets of environments in $\mathit{VMem} \times \mathit{AMem}$, and satisfying the properties shown in Figure 5.3.

$$\begin{aligned} \gamma(d_1 \sqcap d_2) &\supseteq \gamma(d_1) \cap \gamma(d_2) \\ \gamma(d_1 \sqcup d_2) &\supseteq \gamma(d_1) \cup \gamma(d_2) \\ \gamma([\![x{:=}e]\!]^\sharp(d)) &\supseteq \\ &\quad \{\langle \rho_{\mathsf{v}}[x \mapsto\ v], \rho_{\mathsf{a}} \rangle \mid \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}} \rangle \in \gamma(d) \\ &\qquad \wedge\ \ v \in [\![e]\!]_{\langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}} \rangle}\} \\ \gamma([\![x[e_1]{:=}e_2]\!]^\sharp(d)) &\supseteq \\ &\quad \{[\rho_{\mathsf{v}}, [\rho_{\mathsf{a}} \mid x \to [\rho_{\mathsf{a}}\ a \mid v_1 \to v_2]] \mapsto] \rho = \langle \rho_{\mathsf{v}}, \rho_{\mathsf{a}} \rangle \in \gamma(d) \\ &\qquad \wedge\ \ v_1 \in [\![e_1]\!]_\rho\ \wedge\ \ v_2 \in [\![e_2]\!]_\rho\} \\ \gamma(\mathit{assume}^\sharp(t)) &\supseteq \{\rho\ \mid\ [\![t]\!]_\rho\} \end{aligned}$$

Fig. 5.3. Requirements over concretization function γ

We define the abstract test $[\![t]\!]^\sharp : \mathbb{D} \to \mathbb{D}$ of a guard $t \in \mathit{Guard}$ by $[\![t]\!]^\sharp(l^\sharp) = \mathit{assume}^\sharp(t) \sqcap l^\sharp$.

Source Code Analysis

The source code analysis is specified by abstract transfer functions that map elements of the abstract domain into elements of the abstract domain.

Definition 5.1 (Abstract Domain for High-Level). *A result of the analysis for the source program P is described by a mapping Loc in the lattice*

$$State^\sharp = \mathbf{Lab} \to \mathbb{D} \ .$$

$$\frac{}{Loc \vdash \{Loc(i)\}\ [\texttt{Skip}]^i\ \{Loc(i)\}} \qquad \frac{}{Loc \vdash \{Loc(i)\}\ [\texttt{return}\ e]^i\ \{\bot\}}$$

$$\frac{}{Loc \vdash \{Loc(i)\}\ [x{:=}e]^i\ \{[\![x{:=}e]\!]^\sharp(Loc(i))\}}$$

$$\frac{}{Loc \vdash \{Loc(i)\}\ [a[x]{:=}y]^i\ \{[\![a[x]{:=}y]\!]^\sharp(Loc(i))\}}$$

$$\frac{Loc \vdash \{[\![t]\!]^\sharp(Loc(i))\}\ s_1\ \{l_1^\sharp\} \qquad Loc \vdash \{[\![\neg t]\!]^\sharp(Loc(i))\}\ s_2\ \{l_2^\sharp\}}{Loc \vdash \{Loc(i)\}\ \texttt{if}\ [t]^i\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \{l_1^\sharp \sqcup l_2^\sharp\}}$$

$$\frac{Loc \vdash \{[\![t]\!]^\sharp(Loc(i))\}\ s\ \{l^\sharp\} \qquad l^\sharp \sqsubseteq Loc(i)}{Loc \vdash \{Loc(i)\}\ \texttt{while}\ [t]^i\ \texttt{do}\ s\ \{[\![\neg t]\!]^\sharp(Loc(i))\}}$$

$$\frac{Loc \vdash \{l^\sharp\}\ s_1\ \{l_1^\sharp\} \qquad Loc \vdash \{l_1^\sharp\}\ s_2\ \{l_2^\sharp\}}{Loc \vdash \{l^\sharp\}\ s_1\,;\,s_2\ \{l_2^\sharp\}}$$

$$\frac{Loc \vdash \{\top\}\ P\ \{l^\sharp\}}{Loc \vdash P}$$

Fig. 5.4. Definition of the constraint system for the source code analysis.

Definition 5.2 (Solution). *A mapping Loc for the source program P is a solution of the analysis if it verifies the constraint system defined in Figure 5.4, i.e., $Loc \vdash P$ holds.*

Bytecode Analysis

As for the source code analysis, the bytecode analysis is defined by abstract transfer functions that map abstract states into abstract states. In this case, the abstract states are pairs of the form $(s^\sharp, l^\sharp)$ where $l^\sharp$ is an element of the abstract domain, and the list of symbolic expressions $s^\sharp$ abstracts the operand stack. The symbolic abstract domain for stacks is $Expr^\star$, where for any set

A, $A^\star$ denotes the domain of lists with elements in A. The set of variables considered by the bytecode analysis is the same as in the source code analysis.

Definition 5.3 (Bytecode Abstract Domain). *A result of the analysis for $\dot{p}$ is described by a mapping $l\dot{o}c$ in the lattice*

$$\dot{state}^\sharp = \mathbf{Lab} \to (Expr_L^\star \times \mathbb{D}) \ .$$

An analysis result is a solution of the analysis if it satisfies the constraint system associated to each program. The constraint system is defined in Figure 5.5. For instructions other than branching or return instructions, the constraint is defined by partial transfer functions in $Expr^\star \times \mathbb{D} \rightharpoonup (Expr^\star \times \mathbb{D})$, most of them defined as a symbolic execution affecting the abstract representation of the operand stack.

$instr$	$\dot{f}_{instr}$
`prim` op	$(e_1 :: e_2 :: s^\sharp, l^\sharp) \to (\llcorner e_1 \ \underline{op} \ e_2 \lrcorner :: s^\sharp, l^\sharp)$
`push` n	$(s^\sharp, l^\sharp) \to (n :: s^\sharp, l^\sharp)$
`load` r	$(s^\sharp, l^\sharp) \to (\llcorner r \lrcorner :: s^\sharp, l^\sharp)$
`store` r	$(e :: s^\sharp, l^\sharp) \to (s^\sharp[?/r], [\![r{:=}e]\!]^\sharp(l^\sharp))$
`aload` a	$(e :: s^\sharp, l^\sharp) \to (\llcorner a[e] \lrcorner :: s^\sharp, l^\sharp)$
`astore` a	$(e_1 :: e_2 :: s^\sharp, l^\sharp) \to (s^\sharp[?/a], [\![a[e_1]{:=}e_2]\!]^\sharp(l^\sharp))$
`nop`	$(s^\sharp, l^\sharp) \to (s^\sharp, l^\sharp)$

$$\frac{\mathsf{Instr} \notin \{\ \texttt{jmp}\ i', \texttt{cjmp} \bowtie i', \texttt{return}\} \qquad \dot{f}_{instr}(l\dot{o}c(i)) \sqsubseteq l\dot{o}c(i+1)}{l\dot{o}c \vdash i : \mathsf{Instr}}$$

$$\frac{}{l\dot{o}c \vdash i : \texttt{return}} \qquad \frac{l\dot{o}c(i) \sqsubseteq l\dot{o}c(j)}{l\dot{o}c \vdash i : \texttt{jmp}\ j}$$

$$\frac{\begin{array}{c} l\dot{o}c(i) = (e_1 :: e_2 :: s^\sharp, l^\sharp) \\ (s^\sharp, [\![\neg(e_1 \bowtie e_2)]\!]^\sharp(l^\sharp)) \sqsubseteq l\dot{o}c(i+1) \qquad (s^\sharp, [\![e_1 \bowtie e_2]\!]^\sharp(l^\sharp)) \sqsubseteq l\dot{o}c(j) \end{array}}{l\dot{o}c \vdash i : \texttt{cjmp} \bowtie j}$$

$$\frac{\top \sqsubseteq l\dot{o}c(0) \qquad \forall i \in \mathsf{dom}(\dot{p}).\ \ l\dot{o}c \vdash i : \dot{p}[i]}{l\dot{o}c \vdash \dot{p}}$$

Fig. 5.5. Definition of the constraint system for the bytecode analysis.

Definition 5.4 (Solution). *A mapping $l\dot{o}c$ for the bytecode program $\dot{p}$ is a solution of the analysis if it satisfies the constraint system of Figure 5.5, i.e., if $l\dot{o}c \vdash \dot{p}$ holds.*

Preservation of Solutions

We define first the compilation of a source code analysis solution and then show that it is a solution for the bytecode analysis. For notational convenience, we denote by $\dot{f}_{s_1;\cdots;s_n}(s^\sharp, l^\sharp)$ the composition $\dot{f}_{s_n}(\cdots(\dot{f}_{s1}(s^\sharp, l^\sharp))\cdots)$, where $s_1;\cdots;s_n$ is a sequence of bytecode instructions. Let $\mathsf{succ}(l)$ denote the set of successors of a label l, e.g., $\mathsf{succ}(l) = \emptyset$ and $\mathsf{succ}(l) = \{l+1, l'\}$ for $\dot{p}[l] = \texttt{return}$ and $\dot{p}[l] = \texttt{cjmp} \bowtie l'$, respectively. The set $\mathsf{pred}(l)$ is defined as $\{l' \mid l \in \mathsf{succ}(l')\}$.

Remark 5.5. For each bytecode program $\dot{p}$, we can extract from the previous constraint system a set of transfer functions $(\dot{g}_{i,j})_{(i,j)\in\mathbf{Lab}^2}$ s.t. $\dot{loc} \vdash \dot{p}$ if and only if $\bigsqcup_{k'\in\mathsf{pred}(k)} \dot{g}_{k',k}(\dot{loc}(k')) \sqsubseteq \dot{loc}(k)$ for all $k \in \mathsf{dom}(\dot{p})$.

We can extend a partial function $\dot{loc}_{\text{partial}} \in \dot{state}^\sharp$ to a total function $\dot{loc}$ on $\mathsf{dom}(\dot{p})$ if we set $\dot{loc}(k)$ equal to:

$$\begin{array}{l}\text{if } k \in \mathsf{dom}(\dot{loc}_{\text{partial}}) \text{ then } \dot{loc}_{\text{partial}}(k)\\ \text{else if } k \in \mathsf{dom}(\dot{p}) \text{ then } \bigsqcup_{k'\in\mathsf{pred}(k)} \dot{g}_{k',k}(\dot{loc}(k'))\\ \text{else undef}\end{array}$$

This definition only makes sense if, by considering the control flow graph of $\dot{p}$ whose edges are $\{(i,j) | i \in \mathsf{dom}(\dot{p}) \wedge j \in \mathsf{succ}(i)\}$, every loop contains a label in $\mathsf{dom}(\dot{loc}_{\text{partial}})$. We refer to the function $\dot{loc}$ as the *completion of* $\dot{loc}_{\text{partial}}$.

Definition 5.6 (Compiled analysis results). *Given an analysis result Loc for the program P, an analysis result compiled from Loc is the completion of the function* $\dot{loc}_{partial}$ *defined on each* $k \in \mathsf{dom}(Loc)$ *by* $\dot{loc}_{partial}(k) = ([\,], Loc(k))$.

This definition can be shown to be well defined from the facts that *Loc* annotates every loop in P and that each loop in the control flow graph of $\dot{p}$ contains a label of a loop in P.

Lemma 5.7. *Let* $\dot{p}_1$, $\dot{p}_2$ *and* e *such that* $\dot{p} = \dot{p}_1; l : [\![e]\!]_{\mathsf{e}}; l' : \dot{p}_2$. *Then,* $\dot{loc}(l') = f_{i_1;\ldots;i_k}(s^\sharp, l^\sharp) = (e :: s^\sharp, l^\sharp)$ *where* $(s^\sharp, l^\sharp) = \dot{loc}(l)$ *and* $[i_1;\ldots;i_k] = [\![e]\!]_{\mathsf{e}}$.

Proof. We prove the lemma by structural induction over the expression e.

Lemma 5.8. *Let* $\dot{p}_1$, $\dot{p}_2$ *and* e *such that* $\dot{p} = \dot{p}_1; k_1 : [\![e]\!]_{\mathsf{e}}; k_2 : \dot{p}_2$. *Then*

$$\forall k \in [k_1, k_2),\ \dot{loc} \vdash k : \dot{p}[k].$$

Proof. We shall prove it by induction over expression e. From the definition of the compiler one sees that $k_2 \notin \mathsf{dom}(Loc)$.

Case $e = n$. *In this case* $[\![e]\!]_{\mathsf{e}} = \texttt{push}\ n$, $[k_1, k_2) = \{k_1\}$ *and since* $\mathsf{pred}(k_1 + 1) = \{k_1\}$, $\dot{loc}(k_1{+}1) = \dot{f}_{\texttt{push}\ n}(\dot{loc}(k_1))$, *which implies that* $\dot{loc} \vdash k_1 : \texttt{push}\ n$.

Case $e = x$. *We have* $[\![e]\!]_{\mathrm{e}} = \texttt{load}\ x$, $[k_1, k_2) = \{k_1\}$ *and since* $\mathsf{pred}(k_1+1) = \{k_1\}$, $l\dot{o}c(k_1+1) = \dot{f}_{\texttt{load}\ x}(l\dot{o}c(k_1))$, *which implies that* $l\dot{o}c \vdash k_1 : \texttt{load}\ x$.
Case $e = a[e']$. *Here* $[\![e]\!]_{\mathrm{e}} = [\![e']\!]_{\mathrm{e}};\ \texttt{aload}\ a$. *Let* $k' = k_1 + |[\![e']\!]_{\mathrm{e}}|$. *By induction hypothesis we have that* $\forall k \in [k_1, k'),\ l\dot{o}c \vdash k : \dot{p}[k]$ *and since* $\mathsf{pred}(k'+1) = \{k'\}$ *then* $\dot{f}_{\texttt{aload}\ a}(l\dot{o}c(k')) = l\dot{o}c(k'+1)$ *which implies that* $l\dot{o}c \vdash k' : \texttt{aload}\ a$.
Case $e = e_1 \texttt{op}\ e_2$. *This give* $[\![e]\!]_{\mathrm{e}} = [\![e_2]\!]_{\mathrm{e}};\ [\![e_1]\!]_{\mathrm{e}};\ \texttt{prim op}$. *Let* $k'' = k_1 + |[\![e'']\!]_{\mathrm{e}}|$ *and* $k' = k'' + |[\![e']\!]_{\mathrm{e}}|$. *By induction hypothesis,* $\forall k \in [k_1, k'') \cup [k'', k'),\ l\dot{o}c \vdash k : \dot{p}[k]$ *and since the only predecessor of* $k_1 + 1$ *is* k, $l\dot{o}c(k'+1) = \dot{f}_{\texttt{prim op}}(l\dot{o}c(k'))$, *which means that* $l\dot{o}c \vdash k' : \texttt{prim op}$.

The following lemma states the main result of this section: compilation preserves analysis solutions.

Lemma 5.9. *If Loc is s.t.* $Loc \vdash P$, *then the analysis result* $l\dot{o}c$ *compiled from Loc is s.t.* $l\dot{o}c \vdash \dot{p}$, *i.e., it is a solution of the bytecode analysis.*

Proof. Suppose that $\dot{p} = \dot{p}_1; k_1 : [\![s]\!]; k_2 : \dot{p}_2$ *and that there exists* $l^\sharp$ *such that* $Loc \vdash \{Loc(k_1)\}\ s\ \{l^\sharp\}$ *and* $([], l^\sharp) \sqsubseteq l\dot{o}c(k_2)$. *We shall prove that* $\forall k \in [k_1, k_2),\ l\dot{o}c \vdash k : \dot{p}[k]$. *In order to do that, we proceed by induction over statement s. In this proof we omit the calculus of the primed labels.*

Case $s = [\texttt{Skip}]^{k_1}$. *We have* $[\![s]\!] = \texttt{nop}$. *Since*

$$\begin{aligned}
\dot{f}_{\texttt{nop}}(l\dot{o}c(k_1)) &= l\dot{o}c(k_1) \\
&= ([], Loc(k_1)) \\
&= ([], F_{\texttt{Skip}}(l\dot{o}c(k_1))) \\
&= ([], l^\sharp) \\
&\sqsubseteq l\dot{o}c(k_2) \\
&= l\dot{o}c(k_1+1),
\end{aligned}$$

then $l\dot{o}c \vdash k_1 : \texttt{nop}$.
Case $s = [x{:=}e]^{k_1}$. *Here* $[\![s]\!] = [\![e]\!]_{\mathrm{e}}; k_1' : \texttt{store}\ x$. *By Lemma 5.8,* $\forall k \in [k_1, k_1'),\ l\dot{o}c \vdash k : \dot{p}[k]$ *and since*

$$\begin{aligned}
\dot{f}_{\texttt{store}\ x}(l\dot{o}c(k_1')) &= \dot{f}_{\texttt{store}\ x}(([e], Loc(k_1'))) \\
&= ([], [\![x{:=}e]\!]^\sharp(Loc(k_1))) \\
&= ([], F_{x:=e}(l\dot{o}c(k_1))) \\
&= ([], l^\sharp) \\
&\sqsubseteq l\dot{o}c(k_2) \\
&= l\dot{o}c(k_1'+1),
\end{aligned}$$

also $l\dot{o}c \vdash k_1' : \texttt{store}\ x$.
Case $s = [a[e_1]{:=}e_2]^{k_1}$. $[\![s]\!] = [\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}}; k_1' : \texttt{astore}\ a$.
By Lemma 5.8, $\forall k \in [k_1, k_1'),\ l\dot{o}c \vdash k : \dot{p}[k]$ *and since*

$$\begin{aligned}\dot{f}_{\texttt{astore } a}(\dot{loc}(k_1')) &= \dot{f}_{\texttt{astore } a}(([e_1,e_2], Loc(k_1')))\\ &= ([], [\![a[e_1]{:=}e_2]\!]^\sharp(Loc(k_1)))\\ &= ([], F_{a[e_1]:=e_2}(\dot{loc}(k_1)))\\ &= ([], l^\sharp)\\ &\sqsubseteq \dot{loc}(k_2)\\ &= \dot{loc}(k_1'+1),\end{aligned}$$

also $\dot{loc} \vdash k_1' : \texttt{astore } a$.

Case $s = s_1; s_2$. *In this case* $[\![s]\!] = [\![s_1]\!]; k_1' : [\![s_2]\!]$. *By inductive hypothesis we know that* $\forall k \in [k_1, k_1') \cup [k_1', k_2),\ \dot{loc} \vdash k : \dot{p}[k]$.

Case $s = [\texttt{return } e]^{k_1}$. *Here* $[\![s]\!] = [\![e]\!]_{\mathrm{e}}; k_1' : \texttt{return}$. *By Lemma 5.8,* $\forall k \in [k_1, k_1'),\ \dot{loc} \vdash k : \dot{p}[k]$. *Also,* $\dot{loc} \vdash k_1' : \texttt{return}$ *is always true.*

Case $s = \texttt{if } [e_1 \bowtie e_2]^{k_1} \texttt{ then } s_1 \texttt{ else } s_2$. *We have* $[\![s]\!] = [\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}}; k_1' : \texttt{cjmp} \bowtie k_4'; k_2' : [\![s_2]\!]; k_3' : \texttt{jmp } k_2; k_4' : [\![s_1]\!]_{\mathrm{e}}$. *By Lemma 5.8 and inductive hypothesis we know that* $\forall k \in [k_1, k_1') \cup [k_2', k_3') \cup [k_4', k_2),\ \dot{loc} \vdash k : \dot{p}[k]$

By hypothesis we have

$$\begin{gathered}Loc \vdash \{[\![e_1 \bowtie e2]\!]^\sharp(Loc(k_1))\}\ s_1\ \{l_1^\sharp\}\\ \text{and } Loc \vdash \{[\![\neg(e_1 \bowtie e2)]\!]^\sharp(Loc(k_1))\}\ s_2\ \{l_2^\sharp\}\end{gathered}$$

and $l^\sharp = l_1^\sharp \sqcup l_2^\sharp$.

It can be proved that for every judgment of the form $Loc \vdash \{d_1^\#\}\ s\ \{d_2^\#\}$ *we have* $d_1^\# = Loc(init(s))$. *Therefore,* $\dot{loc}(k_4') = ([], Loc(k_4')) = ([], [\![e_1 \bowtie e_2]\!]^\sharp(Loc(k_1)))$ *and* $\dot{loc}(k_2') = ([], Loc(k_2')) = ([], [\![\neg(e_1 \bowtie e_2)]\!]^\sharp(Loc(k_1)))$. *Additionally,* $Loc(k_1') = ([e_1, e_2], Loc(k_1))$ *by Lemma 5.7. Thus,* $\dot{loc} \vdash k_1' : \texttt{cjmp} \bowtie k_4'$.

One can show that for all s *s.t.* $\dot{p} = \dot{p}_1; k : [\![s]\!]; k' : \dot{p}_2$ *and* $k' \notin \mathsf{dom}(Loc)$ *and* $Loc \vdash \{Loc(init(s))\}\ s\ \{l^\sharp\}$, $\dot{loc}(k') \sqsubseteq ([], l^\sharp)$. *Since* $k_3' \notin \mathsf{dom}(Loc)$, $\dot{loc}(k_3') \sqsubseteq ([], l_2^\sharp)$. *Also,* $([], l_2^\sharp) \sqsubseteq ([], l^\sharp) \sqsubseteq \dot{loc}(k_2)$. *Then,* $\dot{loc}(k_3') \sqsubseteq \dot{loc}(k_2)$ *implies* $\dot{loc} \vdash k_3' : \texttt{jmp } k_2$, *which completes the proof for this case.*

Case $s = \texttt{while } [e_1 \bowtie e_2]^{k_1} \texttt{ do } s'$. $[\![s]\!] = [\![e_2]\!]_{\mathrm{e}}; [\![e_1]\!]_{\mathrm{e}}; k_1' : \texttt{cjmp} \bowtie k_3'; k_2' : \texttt{jmp } k_2; k_3' : [\![s']\!]; k_4' : \texttt{jmp } k_1$. *By Lemma 5.8 and Induction Hypothesis we know that* $\forall k \in [k_1, k_1') \cup [k_3', k_4'),\ \dot{loc} \vdash k : \dot{p}[k]$.

Using Lemma 5.7,

$$\dot{loc}(k_1') = ([e_1, e_2], Loc(k_1)) \tag{5.1}$$

Since $k_2' \notin Loc$ *and* $\mathsf{pred}(k_2') = \{k_1'\}$,

$$\begin{aligned}\dot{loc}(k_2') &= \dot{g}_{k_1', k_2'}(\dot{loc}(k_1'))\\ &= ([], [\![\neg(e_1 \bowtie e_2)]\!]^\sharp(Loc(k_1)))\end{aligned} \tag{5.2}$$

Given that $Loc \vdash \{[\![e_1 \bowtie e_2]\!]^\sharp(Loc(k_1))\}\ s'\ \{l_{s'}^\sharp\}$ *holds assuming our hypothesis and* $k_3' = init(s')$, *we have that*

$$l\dot{o}c(k_3') = ([], [\![e_1 \bowtie e_2]\!]^\sharp(Loc(k_1))) \tag{5.3}$$

As we said, $l\dot{o}c(k_4') \sqsubseteq l^\sharp_{s'}$ *because* $k_4' \notin Loc$*. This gives us*

$$l\dot{o}c(k_4') \sqsubseteq l\dot{o}c(k_2) \tag{5.4}$$

Also, by hypothesis,

$$[\![\neg(e_1 \bowtie e_2)]\!]^\sharp(Loc(k_1)) \sqsubseteq l\dot{o}c(k_2) \tag{5.5}$$

Then, (5.1), (5.2) and (5.3) implies $l\dot{o}c \vdash k_1' : \texttt{cjmp} \bowtie k_3'$*. (5.2) and (5.5) implies* $l\dot{o}c \vdash k_2' : \texttt{jmp}\ k_2$*, and (5.4) implies* $l\dot{o}c \vdash k_4' : \texttt{jmp}\ k_1$*, which complete the proof for this last case.*

5.3 Preservation of proof obligations

In this section, we define two verification frameworks, for source programs and for unstructured bytecode of the previous sections, respectively. As a specification language we consider first order formulae, namely the domain of assertions $\mathcal{A}$. The validity of an assertion in a particular execution state $\eta \in State^s$ is standard. In particular, an assertion that contains the expression $a[e]$ is invalid in those execution states in which e is out of the bounds of the array a.

We consider as a program specification a tuple $(\varphi, \mathsf{annot}, \psi, \chi)$, where the assertion φ is a precondition, ψ and χ are normal and abnormal postconditions, respectively, and the partial function $\mathsf{annot} : \mathbf{Lab} \rightharpoonup \mathcal{A}$ maps program labels to assertions. The special variable res may only occur in ψ, and φ only refers to variables from V. When specifying a bytecode program, assertions may refer to the special variable s representing the operand stack.

We say that a program satisfies the specification $(\varphi, \mathsf{annot}, \psi, \chi)$, if every execution starting in a state that satisfies φ only reaches normal final states satisfying ψ or abnormal states satisfying χ, and only reaches intermediate l-labeled points satisfying $\mathsf{annot}(l)$. Given a program specification $(\varphi, \mathsf{annot}, \psi, \chi)$, a verification condition generator (VCgen) framework provides a set of sufficient proof obligations that ensures that the program satisfies the specification.

The VCgen defined in this section is hybrid in the sense that it takes advantage of a previously computed analysis to reduce the size of proof obligations. We assume that the result of a relational analysis (Loc and $l\dot{o}c$ for source and bytecode programs, respectively) is given as input to the VCgen. For the abstract domain $\mathbb{D}$, we consider a relation $\models \subseteq \mathbb{D} \times \mathcal{A}$ such that for any guard b and any $d \in \mathbb{D}$, $d \models b$ indicates that the interpretation of the abstract element d ensures the validity of the condition b. For example, when accessing an array in the expression $a[x]$ we shall check that the value of the variable x is within the bounds of the array a. If we instantiate $\mathbb{D}$ with the

domain of convex polyhedra, each element $d \in \mathbb{D}$ represents a set of linear constraints from which we can discover whether the condition $0 \leq x < |a|$ is satisfied.

A further improvement over standard VCgen consists of reusing the result of the analysis to strengthen loop invariants. This technique helps reducing the size of annotations and the burden of interactive specification. To that end, we assume a concretization function $\gamma_{\mathsf{a}} : \mathbb{D} \to \mathcal{A}$ to interpret abstract elements $d \in \mathbb{D}$ as assertions.

VCgen for Source Programs

Consider a specification $(\varphi, \mathsf{annot}, \psi, \chi)$ for the source program P. Throughout this section, we assume that annot sufficiently annotates the program P, that is, for every subprogram $\texttt{while}\ [t]^l\ \texttt{do}\ c$ of P, we have that $l \in \mathsf{dom}(\mathsf{annot})$.

A VCgen for source programs is defined by the set of proof obligations:

$$\mathsf{PO} = \{\varphi \Rightarrow \phi[{}^{\vec{V}}/_{\vec{V}^{old}}]\} \cup \theta$$

where $\langle \phi, \theta \rangle = \mathsf{WP}(P, \psi)$, $\phi[{}^{\vec{V}}/_{\vec{V}^{old}}]$ represents the result of substituting in ϕ any array or scalar variable x^{old} in $V_s^{old} + V_a^{old}$ by x, and the function WP is defined in Figure 5.6. In the figure, the assertion $\mathsf{inB}(e)$ stands for the condition that must satisfy an execution state to ensure that every array access in e is within bounds. For instance, if e does not contain array expressions, $\mathsf{inB}(e)$ is defined as true and $\mathsf{inB}(a[e])$ as $0 \leq e < |a|$. We follow the simplifying assumption that expressions contain no more than one array access. For any array variable a and expressions e_1 and e_2, $\mathsf{upd}(a, e_1, e_2)$ is interpreted as the array a' such that $a'[e]$ is evaluated to e_2 if $e_1 = e$ and to $a[e]$ otherwise. To simplify the presentation of examples, proof obligations for $\texttt{while}$ statements are split into two assertions corresponding to the true and false branches.

The function WP considers the result of the analysis Loc to reduce the size of proof obligations. That is, if the abstract value $Loc(l)$ associated to the program point under consideration indicates that any array access in the statement is within bounds, the returned predicate is simplified by omitting the exceptional postcondition. Consider the program of Figure 5.7. If the analysis is able to compute at label k_1 an abstract value d such that $d \models 0 \leq i < |A|$, the WP function will return the assertion $\mathsf{upd}(A, i, A[0])[i + 1 - 1] = A[0]$, which together with the loop invariant at label k yields the proof obligation

$$A[i-1] = 0 \wedge \boxed{0 \leq i \leq |A|} \Rightarrow i < |A| \Rightarrow \mathsf{upd}(A, i, A[0])[i + 1 - 1] = A[0]$$

where the boxed assertion $\boxed{0 \leq i \leq |A|}$ represents the interpretation of the result of the analysis at the loop entry point.

In contrast, if we do not take advantage of the result of the analysis we are due to prove the equivalent but bigger formula:

$$A[i-1] = 0 \wedge \boxed{0 \leq i \leq |A|} \Rightarrow i < |A| \Rightarrow \\ (0 \leq i < |A| \Rightarrow \mathsf{upd}(A, i, A[0])[i+1-1] = A[0] \wedge \neg(0 \leq i < |A|) \Rightarrow \mathsf{false}) \ .$$

$$\frac{}{\mathsf{WP}([x{:=}e]^l, \phi) = \langle \mathsf{ckB}(e, \phi[{}^{e}\!/_{x}]), \emptyset \rangle} \qquad \frac{}{\mathsf{WP}([\mathtt{return}\ e]^l, \phi) = \langle \mathsf{ckB}(e, \psi[{}^{e}\!/_{\mathsf{res}}]), \emptyset \rangle}$$

$$\frac{}{\mathsf{WP}(\mathtt{Skip}, \phi) = \langle \phi, \emptyset \rangle} \qquad \frac{\mathsf{WP}(c_1, \phi_2) = \langle \phi_1, \theta_1 \rangle \qquad \mathsf{WP}(c_2, \phi) = \langle \phi_2, \theta_2 \rangle}{\mathsf{WP}(c_1; c_2, \phi) = \langle \phi_1, \theta_1 \cup \theta_2 \rangle}$$

$$\frac{}{\mathsf{WP}([a[e_1]{:=}e_2]^l, \phi) = \langle \mathsf{ckB}(e_2, \mathsf{ckB}(a[e_1], \phi[{}^{\mathsf{upd}(a,e_1,e_2)}\!/_{a}])), \emptyset \rangle}$$

$$\frac{\mathsf{WP}(c_1, \phi) = \phi_1, \theta_1 \qquad \mathsf{WP}(c_2, \phi) = \phi_2, \theta_2}{\mathsf{WP}(\mathtt{if}\ [t]^l\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2, \phi) = \langle \mathsf{ckB}(t, t \Rightarrow \phi_1 \wedge \neg t \Rightarrow \phi_2), \theta_1 \cup \theta_2 \rangle}$$

$$\frac{\mathsf{WP}(c, \phi) = \langle \phi_1, \theta \rangle \qquad \Phi = (t \Rightarrow \phi_1) \wedge (\neg t \Rightarrow \phi)}{\mathsf{WP}(\mathtt{while}\ [t]^l\ \mathtt{do}\ c, \phi) = \langle \mathsf{annot}(l), \{\mathsf{annot}(l) \wedge \gamma_{\mathsf{a}}(\mathit{Loc}(l)) \Rightarrow \mathsf{ckB}(t, \Phi)\} \cup \theta_1 \rangle}$$

where the expression $\mathsf{ckB}(e, \varphi)$ stands for φ if $\mathit{Loc}(l) \models \mathsf{inB}(e)$ and for the formula $(\mathsf{inB}(e) \Rightarrow \varphi) \wedge (\neg\mathsf{inB}(e) \Rightarrow \chi)$ otherwise.

Fig. 5.6. Definition of WP function

As can be seen from the definition of WP, proof obligations computed by the hybrid VCGen are of the form $\phi_1 \wedge \boxed{\gamma_{\mathsf{a}}(d)} \Rightarrow \phi_2$, whereas a standard VCgen would output the stronger proof obligation $\phi_1 \Rightarrow \phi_2$. In consequence, one can provide the code with a weaker invariant ϕ_1 as long as the analyzer is able to eventually infer the missing information $\gamma_{\mathsf{a}}(d)$. For instance, for the simple program of Figure 5.7, a standard VCgen will return the invalid proof obligation

$$A[i-1] = A[0] \Rightarrow \neg(i < |A|) \Rightarrow A[|A| - 1] = A[0]$$

for the path that does not enter the loop. It is sufficient to provide a stronger invariant, i.e., to join it to the condition $i \leq |A|$, to prove the program correct. However, as an alternative to increasing the size of the program annotations, assuming the condition $i \leq |A|$ is inferred by the analysis, the hybrid VCgen generates the weaker (and valid) proof obligation

$$A[i-1] = A[0] \wedge \boxed{0 \leq i \leq |A|} \Rightarrow \neg(i < |A|) \Rightarrow A[|A| - 1] = A[0] \ .$$

```
//φ : true,  χ : false
[i := 1]^k0;
//A[i − 1] = A[0]
while [i < |A|]^k do {
   [A[i] := A[0]]^k1;
   [i := i + 1]^k2
}
//A[|A| − 1] = A[0]
...
```

Fig. 5.7. Program example

VCgen for Bytecode Programs

Let $(\varphi, \mathsf{annot}, \psi, \chi)$ be a specification for the bytecode program $\dot{p}$. As with the VCgen for source programs defined above, the precondition φ and the internal annotations $\mathsf{annot}(l)$ are strengthened with the result of the analysis. To that end, we interpret the result of the analysis with the aid of the concretization functions $\gamma_{\mathsf{a}} : \mathbb{D} \to \mathcal{A}$ and $\bar{\gamma}_{\mathsf{a}} : (Expr^{\star} \times \mathbb{D}) \to \mathcal{A}$. A VCgen for bytecode is defined by extracting the set of proof obligations:

$$\mathsf{po} = \{\varphi \Rightarrow \mathsf{wp}_i(0)[{}^{V}/_{V^{\vec{o}ld}}]\} \cup \{\mathsf{annot}(l) \wedge \bar{\gamma}_{\mathsf{a}}(l\dot{o}c(l)) \Rightarrow \mathsf{wp}_i(l) \mid l \in \mathsf{dom}(\mathsf{annot})\}$$

where the predicate transformer $\mathsf{wp}_{\mathcal{L}}$ is shown in Figure 5.8. If the program point is annotated, the function $\mathsf{wp}_{\mathcal{L}}$ returns $\mathsf{annot}(l)$. Otherwise it applies the weakest precondition transformer wp_i, defined in terms of the instruction at program point l, taking as parameters the annotations computed for the successor program points. The definition of $\mathsf{wp}_{\mathcal{L}}$ and wp_i is done by induction along the control flow paths of the program. We say that a program $\dot{p}$ is sufficiently annotated if the control flow graph of the program $\dot{p}$ does not contain unannotated loops. The induction principle following from the definition of sufficiently annotated programs is sufficient to ensure that $\mathsf{wp}_{\mathcal{L}}$ and wp_i are well defined For a list s, $s[0]$ and $s[1]$ represent the first and second element of s, and $\uparrow s$ denotes the result of removing the first element from s.

Preservation of Proof Obligations

Consider the specification $(\varphi, \mathsf{annot}, \psi, \chi)$ for source program P, and assume that annot is a sufficient annotation for P, i.e., every loop is annotated. Let $(\varphi, \mathsf{annot}, \psi, \chi)$ define as well the specification for the bytecode program $\dot{p}$. From previous results [14], we know that if annot is a sufficient annotation for P then it is also a sufficient annotation for the result of the compilation $\dot{p}$. Let Loc be a solution of the analysis for the source program P, and $l\dot{o}c$ a solution

$$
\begin{array}{ll}
\mathsf{wp}_i(l) = \mathsf{wp}_{\mathcal{L}}(l+1)[^{\mathbf{s}[0]\ \mathtt{op}\ \mathbf{s}[1]::\uparrow^2\mathbf{s}}/_{\mathbf{s}}] & \dot{p}[l] = \mathtt{prim}\ \mathtt{op} \\
\mathsf{wp}_i(l) = \mathsf{wp}_{\mathcal{L}}(l+1)[^{v::\mathbf{s}}/_{\mathbf{s}}] & \dot{p}[l] = \mathtt{push}\ v \\
\mathsf{wp}_i(l) = \mathsf{wp}_{\mathcal{L}}(l+1)[^{\mathbf{s}[0],\uparrow\mathbf{s}}/_{x,\mathbf{s}}] & \dot{p}[l] = \mathtt{store}\ x \\
\mathsf{wp}_i(l) = \mathsf{wp}_{\mathcal{L}}(l+1)[^{x::\mathbf{s}}/_{\mathbf{s}}] & \dot{p}[l] = \mathtt{load}\ x \\
\mathsf{wp}_i(l) = \mathsf{ckB}(\mathsf{wp}_{\mathcal{L}}(l+1)[^{\mathsf{upd}(a,\mathbf{s}[0],\mathbf{s}[1]),\uparrow^2\mathbf{s}}/_{a,\mathbf{s}}]) & \dot{p}[l] = \mathtt{astore}\ a \\
\mathsf{wp}_i(l) = \mathsf{ckB}(\mathsf{wp}_{\mathcal{L}}(l+1)[^{a[\mathbf{s}[0]]::\uparrow\mathbf{s}}/_{\mathbf{s}}]) & \dot{p}[l] = \mathtt{aload}\ a \\
\mathsf{wp}_i(l) = \mathbf{s}[0] \bowtie \mathbf{s}[1] \Rightarrow \mathsf{wp}_{\mathcal{L}}(l')[^{\uparrow^2\mathbf{s}}/_{\mathbf{s}}] & \dot{p}[l] = \mathtt{cjmp} \bowtie l' \\
\qquad \wedge \neg(\mathbf{s}[0] \bowtie \mathbf{s}[1]) \Rightarrow \mathsf{wp}_{\mathcal{L}}(l+1)[^{\uparrow^2\mathbf{s}}/_{\mathbf{s}}] & \\
\mathsf{wp}_i(l) = \mathsf{wp}_{\mathcal{L}}(l') & \dot{p}[l] = \mathtt{jmp}\ l' \\
\mathsf{wp}_i(l) = \mathsf{wp}_{\mathcal{L}}(l+1) & \dot{p}[l] = \mathtt{nop} \\
\mathsf{wp}_i(l) = \psi[^{\mathbf{s}[0]}/_{\mathsf{res}}] & \dot{p}[l] = \mathtt{return}
\end{array}
$$

where $\mathsf{ckB}(\psi)$ stands for ψ if $l\dot{o}c(l) \models \mathsf{inB}(x[\mathbf{s}[0]])$ and $\mathsf{inB}(x[\mathbf{s}[0]]) \Rightarrow \psi \wedge \neg\mathsf{inB}(x[\mathbf{s}[0]]) \Rightarrow \chi$ otherwise.

$$
\mathsf{wp}_{\mathcal{L}}(l) = \begin{cases} \mathsf{annot}(l) \text{ if } l \in \mathsf{dom}(\mathsf{annot}) \\ \mathsf{wp}_i(l) \qquad\qquad \text{otherwise} \end{cases}
$$

Fig. 5.8. VCgen for bytecode programs

$$
\begin{array}{ll}
k_0\mathtt{push}\ 1 & \mathtt{load}\ i \\
\quad \mathtt{store}\ i & \mathtt{prim}\ + \\
\quad k\mathtt{jmp}\ k' & \mathtt{store}\ i \\
k_1\mathtt{push}\ 0 & k'\mathtt{push}\ |A| \\
\quad \mathtt{aload}\ A & \mathtt{load}\ i \\
\quad \mathtt{load}\ i & \mathtt{cjmp} < k_1 \\
\quad \mathtt{astore}\ A & k''.:. \\
k_2\mathtt{push}\ 1 &
\end{array}
$$

Fig. 5.9. Program example

of the analysis for the bytecode program $\dot{p}$, compiled from *Loc* as described in Section 5.2.

We assume that the concretization functions satisfy the property $\bar{\gamma}_{\mathsf{a}}([], d) = \gamma_{\mathsf{a}}(d)$, so that the interpretation of abstract analysis results for the source and bytecode side coincide (recall that by definition $l\dot{o}c(l) = ([\,], Loc(l))$ for every l in $\mathsf{dom}(Loc)$). In addition, for any expression e and any $d \in \mathbb{D}$, if e does not contain array expressions, i.e., $\mathsf{inB}(e) = \mathsf{true}$, then $d \models \mathsf{inB}(e)$.

The following auxiliary result about the compilation of expressions is helpful to prove the preservation of proof obligations:

Lemma 5.10. *Assume that $\dot{p}$ is of the form $\dot{p}_1 :: l_1 : \llbracket e \rrbracket_{\mathsf{e}} :: l_2 : \dot{p}_2$. Then $\mathsf{wp}_i(l_1)$ is equal to $\mathsf{wp}_i(l_2)[^{e::\mathbf{s}}/_{\mathbf{s}}]$ if $l\dot{o}c(l_1) \models \mathsf{inB}(e)$ and equal to $\mathsf{inB}(e) \Rightarrow \mathsf{wp}_i(l_2)[^{e::\mathbf{s}}/_{\mathbf{s}}] \wedge \neg\mathsf{inB}(e) \Rightarrow \chi$ otherwise.*

Proof. The result holds under the assumption stated before that the expression e contains at most one variable access. Otherwise, the syntactic equality of predicates does not hold, but it is straightforward to show a logical equivalence. The proof proceeds by structural induction on the expression e.

The coincidence of the sets of proof obligations PO and po is stated in the following lemma, from the fact that the bytecode program $\dot{p}$ is the result of compiling the source program P.

Proposition 5.11. *For every subprogram c of P, proof obligations corresponding to the subprogram c are equal to the proof obligations in $\dot{p}$ that correspond to the subsequence $\llbracket c \rrbracket$.*

Proof. Assume $\dot{p}$ is of the form $\dot{p}_1 :: l{:}\llbracket c \rrbracket_{\mathsf{e}} :: l'{:}\dot{p}_2$. Let $\langle \phi, \theta \rangle = \mathsf{WP}(c, \mathsf{wp}(l'))$ then one can prove by structural induction on c that $\mathsf{wp}(l) = \phi$ and that θ is equal to

$$\{\mathsf{annot}(k) \wedge \bar{\gamma}_{\mathsf{a}}(l\dot{o}c(k)) \Rightarrow \mathsf{wp}_i(k) \mid k \in \mathsf{dom}(\mathsf{annot}) \cap \mathbf{Lab}_c\},$$

where $\mathbf{Lab}_c$ denotes the set of labels in the statement c.

Consider, the bytecode program of Figure 5.9 compiled from the example in Figure 5.7. One can see that the proof obligation at label k is

$$\begin{array}{l} A[i-1] = A[0] \wedge \boxed{0 \leq i \leq |A|} \Rightarrow \\ \quad (i < |A| \Rightarrow (A[i-1] = A[0])[^{\mathsf{upd}(A,i,A[0]),i+1}/_{A,i}]) \wedge \\ \quad (\neg(i < |A|) \Rightarrow A[|A|-1] = A[0]) \end{array}$$

which is equal to the proof obligation at label k for the source program of Figure 5.7.

5.4 From hybrid VCgen to VCgen

In this section, we show a correspondence between the hybrid VCgen for bytecode of previous section with a standard VCgen that does not take advantage of the result of the analysis. More precisely, interpreting the abstract result as logical formulae, we show an equivalence between the proof obligations of both VCgen's. Assuming that the relation $\models$ satisfies a correctness condition, soundness of the hybrid VCgen follows from soundness of the standard VCgen. In addition, soundness of the VCgen for source programs follows if the compiler is semantics preserving.

Given a specification $(\varphi, \widehat{\mathsf{annot}}, \psi, \chi)$ for the bytecode program $\dot{p}$, a non-hybrid VCgen extracts the set of proof obligations:

$$\hat{\mathsf{po}} \cup \{\varphi \Rightarrow \hat{\mathsf{wp}}_i(0)[{}^{V}/_{V^{old}}]\}$$

where $\hat{\mathsf{wp}}_i$ and $\hat{\mathsf{po}}$ are defined in Figure 5.10. To avoid ambiguity, in the sequel we make explicit some parameters needed in the definition of wp_i, $\mathsf{wp}_{\mathcal{L}}$, $\hat{\mathsf{wp}}_i$ and $\hat{\mathsf{wp}}_{\mathcal{L}}$. We write for instance $\hat{\mathsf{wp}}_i(l, \hat{\mathsf{annot}}, \psi, \chi)$ instead of simply $\hat{\mathsf{wp}}_i(l)$.

$$\begin{array}{ll}
\hat{\mathsf{wp}}_i(l) = \mathsf{ckB}(\hat{\mathsf{wp}}_{\mathcal{L}}(l+1)[{}^{[\mathsf{upd}(x,\mathbf{s}[0],\mathbf{s}[1]),\uparrow^2\mathbf{s}}/_{x,\mathbf{s}}]) & \dot{p}[l] = \texttt{astore}\ x \\
\hat{\mathsf{wp}}_i(l) = \mathsf{ckB}(\hat{\mathsf{wp}}_{\mathcal{L}}(l+1)[{}^{[x[\mathbf{s}[0]]::\uparrow\mathbf{s}}/_{\mathbf{s}}]) & \dot{p}[l] = \texttt{aload}\ x \\
\hat{\mathsf{wp}}_i(l) = \mathsf{wp}_i(l) & \text{otherwise}
\end{array}$$

where $\mathsf{ckB}(\psi)$ stands for

$$\mathsf{inB}(x[\mathbf{s}[0]]) \Rightarrow \psi \wedge \neg\mathsf{inB}(x[\mathbf{s}[0]]) \Rightarrow \chi$$

regardless of whether $l\dot{o}c(l) \models \mathsf{inB}(x[\mathbf{s}[0]])$ is satisfied.

$$\hat{\mathsf{wp}}_{\mathcal{L}}(l) = \begin{cases} \mathsf{annot}(l) \text{ if } l \in \mathsf{dom}(\mathsf{annot}) \\ \hat{\mathsf{wp}}_i(l) \qquad\qquad \text{otherwise} \end{cases}$$

$$\hat{\mathsf{po}} = \{\hat{\mathsf{annot}}(l) \Rightarrow \hat{\mathsf{wp}}_i(l) \mid l \in \mathsf{dom}(\hat{\mathsf{annot}})\}$$

Fig. 5.10. Non-hybrid bytecode VCgen

Let $l\dot{o}c$ be a result of the analysis for the bytecode program $\dot{p}$. Consider the specifications $(\varphi, \mathsf{annot}, \psi, \chi)$ and $(\varphi, \hat{\mathsf{annot}}, \psi, \chi)$ for program $\dot{p}$, such that for all l in $\mathsf{dom}(\mathsf{annot})$, $\hat{\mathsf{annot}}(l)$ is defined as $\mathsf{annot}(l) \wedge \bar{\gamma}_{\mathsf{a}}(l\dot{o}c(l))$. We say that the relation $\models \subseteq \mathbb{D} \times \mathit{Guard}$ is valid if for every abstract element $d \in \mathbb{D}$ and $b \in \mathit{Guard}$ we have that $d \models b$ implies the universal validity of $\gamma_{\mathsf{a}}(d) \Rightarrow b$. The result of the analysis $l\dot{o}c$ is said *verifiable* if the set of proof obligations $\mathsf{po}(\mathsf{true}, \bar{\gamma}_{\mathsf{a}} \circ l\dot{o}c, \mathsf{true}, \mathsf{true})$ are provable.

Lemma 5.12. *For every label l in the program $\dot{p}$:*

$$\mathsf{wp}_i(l, \mathsf{annot}, \psi, \chi) \wedge \bar{\gamma}_{\mathsf{a}}(l\dot{o}c(l)) \Rightarrow \hat{\mathsf{wp}}_i(l, \hat{\mathsf{annot}}, \psi, \chi)$$

provided the relation $\models \subseteq \mathbb{D} \times \mathit{Guard}$ is valid, and the analysis $l\dot{o}c$ is verifiable.

Proof. Following the induction principle induced by the definition of sufficiently annotated programs, for every label l we prove the goal above simultaneously with:

$$\mathsf{wp}_{\mathcal{L}}(l, \mathsf{annot}, \psi, \chi) \wedge \bar{\gamma}_{\mathsf{a}}(l\dot{o}c(l)) \Rightarrow \hat{\mathsf{wp}}_{\mathcal{L}}(l, \hat{\mathsf{annot}}, \psi, \chi)$$

The soundness of the VCgen po follows from the following result and the hypothesis that the standard VCgen $\hat{\mathsf{po}}$ is sound:

Proposition 5.13. *The provability of the set of verification conditions in the set* $\hat{\mathsf{po}}(\varphi, \hat{\mathsf{annot}}, \psi, \chi)$ *follows from the provability of those in the set* $\mathsf{po}(\varphi, \mathsf{annot}, \psi, \chi)$.

Consider for instance the sequence of bytecode in Figure 5.9. Recall that annot is defined as $A[i-1] = A[0]$ and $A[|A|-1] = A[0]$ in k and k'', respectively. Let $\hat{\mathsf{annot}}$ be defined by strengthening annot with the result of the analysis, i.e., $\hat{\mathsf{annot}}(k) = \mathsf{annot}(k) \wedge 0 \leq i \leq |A|$ (we can let $\hat{\mathsf{annot}}(k'') = \mathsf{annot}(k'')$). Let Ψ be the weakest precondition computed by the non hybrid VCgen at label k_1:

$$0 \leq i < |A| \Rightarrow (\mathsf{upd}(A, i, A[0])[i+1-1] = A[0] \wedge \ 0 \leq i+1 \leq |A|)$$
$$\wedge \neg(0 \leq i < |A|) \Rightarrow \mathsf{false}$$

which, from Lemma 5.12 is implied by the hybrid wp and the result of the analysis, i.e., by

$$\mathsf{upd}(A, i, A[0])[i+1-1] = A[0] \wedge \boxed{0 \leq i < |A|} \ .$$

As stated in Proposition 5.13, if the proof obligations returned by the hybrid VCgen are valid, and assuming the analysis is verifiable, we have that

$$A[i-1] = A[0] \wedge \boxed{0 \leq i \leq |A|} \Rightarrow i < |A| \Rightarrow \mathsf{upd}(A, i, A[0])[i+1-1] = A[0]$$

and

$$0 \leq i \leq |A| \Rightarrow i < |A| \Rightarrow 0 \leq i < |A|$$

are provable. Then, it follows that the verification condition returned by the standard VCgen

$$A[i-1] = A[0] \wedge 0 \leq i \leq |A| \Rightarrow i < |A| \Rightarrow \Psi$$

is provable.

The above results state that hybrid verification methods can be mapped to standard verification methods. In the context of Proof Carrying Code, one would like to establish the stronger result that hybrid certificates can be compiled into standard certificates. It is in fact possible to prove such a result, using the framework of [10]. However, the compilation of hybrid certificates into standard certificates requires using a certifying analyzer, that generates automatically logical proofs of correctness of the results of the analysis. While it is possible to avoid hybrid methods altogether, e.g., to rely on standard Proof Carrying Code architectures, hybrid methods are beneficial both for the code producer because they reduce significantly the number of proof obligations required to certify code, and for the code consumer, because they yield certificates that are more compact and more efficient to check. Translating certificates of proof obligations from a hybrid to a standard VCgen is interesting to complete a certificate translation process [7] in which original proof obligations are generated by a hybrid VCgen, but in which the targeted Trusted Computed Base has no support for hybrid certificates.

6

Certified Analysis in Hierarchical Memory Models

Parallel programming languages are gradually abandoning the traditional memory model, in which memory is viewed as a flat and uniform structure, in favor of a hierarchical memory model [2, 28, 36], which considers a tree of memories with different bandwidth and latency characteristics. Thus, programming languages for hierarchical memories are designed to exploit the memory hierarchy and are used for programs that require intensive computations on large amounts of data. Languages for hierarchical memories differ from general-purpose concurrent languages in their intent, and in their realization; in particular, such languages are geared towards deterministic programs and do not include explicit primitives for synchronization (typically programs will proceed by dividing computations between a number of cooperating subtasks, that operate on disjoints subsets of the memory).

The purpose of this chapter is to show for a specific example that existing analysis and verification methods can be adapted to hierarchical languages and are tractable. We focus on the Sequoia programming language [30, 37, 35]. Our methods encompass the usual automated and interactive verification techniques (static analyses and program logics) as well as methods to transform correctness proofs along program optimizations, which are of interest in the context of Proof Carrying Code [49]. In summary, the main technical contributions are: i) a generic, sound, compositional proof system to reason about Sequoia programs (Sect. 6.2.1); ii) a sound program logic derived as an instance of the generic proof system (Sect. 6.2.3); iii) algorithms that transform proofs of correctness along with program optimizations such as SPMD distribution or grouping of tasks [37] (Sect. 6.3).

6.1 A primer on Sequoia

Sequoia [30, 37, 35] is a language for developing portable and efficient parallel programs for hierarchical memories. It is based on a small set of constructs that control essential aspects of programming over a hierarchical memory,

such as communication, memory movement, and computation. Computations are organized into self-contained units, called tasks. Tasks can be executed in parallel at the same level of the memory hierarchy, on a dedicated address space, and may rely on subtasks for performing computations; in this case, each subtask can operate on a lower level (and in practice smaller and faster) fragment of the hierarchical memory (i.e., a subtree).

Hierarchical memory.

A hierarchical memory is a tree of memory modules, i.e., of partial functions from a set $\mathcal{L}$ of locations to a set $\mathcal{V}$ of values. In our setting, values are either integers ($\mathbb{Z}$) or booleans ($\mathbb{B}$). Besides, locations are either scalar variables, or arrays elements of the form $A[i]$ where A is an array and i is an index. The set of scalar variables is denoted by $\mathcal{N}_\mathcal{S}$ and the set of array variables is denoted by $\mathcal{N}_\mathcal{A}$. The set of variable names is $\mathcal{N} = \mathcal{N}_\mathcal{S} \cup \mathcal{N}_\mathcal{A}$.

Definition 6.1 (States). *The set $\mathcal{M} = \mathcal{L} \rightharpoonup \mathcal{V}$ of memory modules is defined as the set of partial functions from locations to values. A memory hierarchy representing the machine structure is a memory tree defined as:*

$$\mathcal{T} ::= \langle \mu, \mathcal{T}_1, \ldots, \mathcal{T}_k \rangle \quad k \geq 0, \mu \in \mathcal{M} \ .$$

Intuitively, $\langle \mu, \vec{\tau} \rangle \in \mathcal{T}$ represents a memory tree with root memory μ and a possible empty sequence $\vec{\tau}$ of child subtrees. The semantics of programs is given using an operator, $+_\mu : \mathcal{M} \times \mathcal{M} \to \mathcal{M}$, indexed by a memory $\mu \in \mathcal{M}$, such that:

$$(\mu_1 +_\mu \mu_2)x = \begin{cases} \mu_1 x & \text{if } \mu_2 x = \mu x, \text{ else} \\ \mu_2 x & \text{if } \mu_1 x = \mu x \\ \text{undefined} & \text{otherwise} \ . \end{cases}$$

Note that the operator $+_\mu$ is partial, and the result is undefined if both μ_1 and μ_2 modify the same variable.

The operator $+_\mu$ is generalized over memory hierarchies in $\mathcal{T}$ and sequences $\vec{\mu} \in \mathcal{M}^\star$, where $\sum^\mu_{1 \leq i \leq n} \mu_i = (((\mu_1 +_\mu \mu_2) +_\mu \mu_3) +_\mu \ldots +_\mu \mu_n)$.

To give an intuition on the operator $+$, consider a memory μ where a task G is to be executed, and assume that G is divided in parallel subtasks $G_0, \ldots, G_n$ that operates on the memory μ. Each subtask G_i executes in its own local copy of the initial memory μ, returning μ_i as final memory. After every subtask G_i has finished execution, the memory state after the execution of G must reflect the changes of the local memory μ_i. Then, the resulting memory after executing G is defined as $\sum^\mu_{0 \leq i \leq n} \mu_i$. En general, subtasks that execute in parallel are intended to operate on pairwise disjoint fragments of the memory μ. For this reason, from the definition of the operator $+$, one can notice that, upon termination of the parallel subtasks, the result $\sum^\mu_{0 \leq i \leq n} \mu_i$ is well defined.

Syntax.

Sequoia features the usual constructions as well as specific constructs for parallel execution, for spawning new subtasks, and for grouping computations.

Definition 6.2 (Sequoia Programs). *The set of programs is defined by the following grammar:*

$$\begin{aligned} G ::= {} & \mathsf{Copy}^{\uparrow}(\vec{A},\vec{B}) \mid \mathsf{Copy}^{\downarrow}(\vec{A},\vec{B}) \mid \mathsf{Copy}(\vec{A},\vec{B}) \\ & \mid \mathsf{Kernel}\langle A = f(B_1,\ldots,B_n)\rangle \mid \mathsf{Scalar}\langle a = f(b_1,\ldots,b_n)\rangle \\ & \mid \mathsf{Forall}\ i = m : n\ \mathsf{do}\ G \mid \mathsf{Group}(H) \mid \mathsf{Exec}_i(G) \\ & \mid \mathsf{If}\ \mathit{cond}\ \mathsf{then}\ G_1\ \mathsf{else}\ G_2 \end{aligned}$$

where a, b are scalar variables, m, n are scalar constants, A, B are array variables, cond is a boolean expression, and H is a dependence graph of programs. We use the operators $\|$ and ; as a syntactic representation (respectively parallel and sequential composition) of the dependence graph composing a Group *task.*

Copy, Kernel, and Scalar statements are consider atomic operations. The constructors Forall and Group start the parallel execution of a set subtasks, and the statement $\mathsf{Exec}_i(G)$ pushes the execution of the subtask G down to the i^{th} memory subtree. A more complete explanation of the semantics of a Sequoia program is given below.

Atomic statements, i.e., Copy, Kernel, and Scalar operations, are given a specific treatment in the proof system; we let atomStmt denote the set of atomic statements. A program G in the dependence graph H is maximal if G is not specified by H to depend on any other program in H.

Semantics.

We now turn to the syntax-directed semantics of programs.

The semantics of a program G is defined by a judgment $\sigma \vdash G \to \sigma'$ where $\sigma, \sigma' \in \mathcal{H}$, and $\mathcal{H} = \mathcal{M} \times \mathcal{T}$. Every $\sigma \in \mathcal{H}$ is a pair $\langle \mu_p, \tau \rangle$ where μ_p is the parent memory and τ is a child memory hierarchy. Abusing nomenclature, we refer to elements of $\mathcal{H}$ as memories. The meaning of such a judgment is that the evaluation of G with initial memory σ terminates with final memory σ'. Note that for a specific architecture, the shape of the memory hierarchy (that is, the shape of the tree structure) is fixed and does not change with the execution of a program.

To manipulate elements in $\mathcal{H}$, we define two functions: $\pi_i : \mathcal{H} \to \mathcal{H}$ that returns the i-th child of a memory, and $\oplus_i : \mathcal{H} \times \mathcal{H} \to \mathcal{H}$ that, given two memories σ_1 and σ_2, replaces the i-th child of σ_1 with σ_2. Formally, they are defined as $\pi_i(\mu_p, \langle \mu, \vec{\tau} \rangle) = (\mu, \tau_i)$ and $(\mu_p, \langle \mu, \vec{\tau} \rangle) \oplus_i (\mu', \tau') = (\mu_p, \langle \mu', \vec{\tau_1} \rangle)$, where $\tau_{1\,i} = \tau'$ and $\tau_{1\,j} = \tau_j$ for $j \neq i$.

Definition 6.3 (Program semantics). *The semantics of a program G is defined by the rules given in Fig. 6.1.*

We briefly comment on the semantics—the omitted rules are either the usual ones (conditionals) or similar to other rules (copy).

The constructs $\mathsf{Copy}^{\downarrow}(\vec{A}, \vec{B})$ and $\mathsf{Copy}^{\uparrow}(\vec{A}, \vec{B})$ are primitives that enable data to migrate along the tree structure, from the parent memory to the root of the child memory hierarchy and conversely; and $\mathsf{Copy}(\vec{A}, \vec{B})$ represents an intra-memory copy in the root of the child memory hierarchy. Only the rule for $\mathsf{Copy}^{\uparrow}$ is shown in Fig. 6.1, since the others are similar.

The constructs $\mathsf{Kernel}\langle A = f(B_1, \ldots, B_n)\rangle$ and $\mathsf{Scalar}\langle a = f(b_1, \ldots, b_n)\rangle$ execute bulk and scalar computations. We implicitly assume in the rules that array accesses are in-bound. If this condition is not met then there is no applicable rule, and the program is stuck. The same happens if, in the rules for Group and Forall, the addition of memories is not defined; that is, the program gets stuck.

The construct $\mathsf{Forall}\ i = m : n\ \mathsf{do}\ G$ executes in parallel $n-m+1$ instances of G with a different value of i, and merges the result. Progress is made only if the instances manipulate pairwise distinct parts of the memory, otherwise the memory after executing the Forall construct is undefined. The rule in Fig. 6.1 considers exclusively the case where $m \leq n$, otherwise the memory hierarchy remains unchanged. The construct $\mathsf{Exec}_i(G)$ spawns a new computation on the i-th subtree of the current memory.

$$\frac{}{\mu_p, \langle \mu, \vec{\tau}\rangle \vdash \mathsf{Copy}^{\uparrow}(\vec{A}, \vec{B}) \rightarrow \mu_p[B \mapsto \mu(A)], \langle \mu, \vec{\tau}\rangle}$$

$$\frac{}{\mu_p, \langle \mu, \vec{\tau}\rangle \vdash \mathsf{Kernel}\langle A = f(B_1, \ldots, B_n)\rangle \rightarrow \mu_p, \langle \mu[A \mapsto f(B_1, \ldots, B_n)], \vec{\tau}\rangle}$$

$$\frac{}{\mu_p, \langle \mu, \vec{\tau}\rangle \vdash \mathsf{Scalar}\langle a = f(b_1, \ldots, b_n)\rangle \rightarrow \mu_p, \langle \mu[a \mapsto f(b_1, \ldots, b_n)], \vec{\tau}\rangle}$$

$$\frac{\begin{array}{c} X \text{ the subset of maximal elements of } H \text{ and } H' = H \setminus X \\ \forall g \in X,\ \mu, \tau \vdash g \rightarrow \mu_g, \tau_g \\ \textstyle\sum_{g \in X}^{\mu, \tau}(\mu_g, \tau_g) \vdash \mathsf{Group}(H') \rightarrow \mu', \tau' \end{array}}{\mu, \tau \vdash \mathsf{Group}(H) \rightarrow \mu', \tau'}$$

$$\frac{\begin{array}{c} \forall j \in [m, n] \neq \emptyset.\ \mu_p, \langle \mu[i \mapsto j], \vec{\tau}\rangle \vdash G \rightarrow \mu_p^j, \langle \mu^j, \vec{\tau}^j\rangle \\ \mu_p, \langle \mu, \vec{\tau}\rangle \vdash \mathsf{Forall}\ i = m : n\ \mathsf{do}\ G \rightarrow \textstyle\sum_{m \leq j \leq n}^{\mu_p, \langle \mu, \tau\rangle}(\mu_p^j, \langle \mu^j[i \mapsto \mu(i)], \vec{\tau}^j\rangle) \\ \pi_i(\mu_p, \langle \mu, \tau\rangle) \vdash G \rightarrow \mu', \tau' \end{array}}{\mu_p, \langle \mu, \vec{\tau}\rangle \vdash \mathsf{Exec}_i(G) \rightarrow (\mu_p, \langle \mu, \tau\rangle) \oplus_i (\mu', \tau')}$$

Fig. 6.1. Sequoia program semantics (excerpt)

Finally, the construct $\mathsf{Group}(H)$ executes the set X of maximal elements of the dependence graph in parallel, and then merges the result before recursively

executing $\mathsf{Group}(H \setminus X)$. A rule not shown in Fig. 6.1 states that if $H = \emptyset$ the state is left unchanged.

6.2 Analyzing and reasoning about Sequoia programs

This section presents a proof system for reasoning about Sequoia programs. We start by generalizing the basics of abstract interpretation to our setting, using a sound, compositional proof system. Then, we define a program logic as an instance of our proof system, and show its soundness.

6.2.1 Program Analysis

We develop our work using a mild generalization of the framework of abstract interpretation, in which abstract elements form a prelattice.

We also have specific operators over the abstract domain for each type of program, as shown below.

Definition 6.4. *An abstract interpretation framework is defined as a tuple* $I = \langle A, T, +_A, \mathsf{weak}, \pi, \oplus, \rho \rangle$ *where:*

- $A = \langle A, \sqsubseteq, \sqsupseteq, \sqcup, \sqcap, \top, \bot \rangle$ *is a prelattice of abstract states;*
- *for each* $s \in \mathsf{atomStmt}$*, a relation* $T_s \subseteq A \times A$*;*
- $+_A : A \times A \to A$*;*
- *for each* $i \in \mathcal{N}_\mathcal{S}$*,* $\mathsf{weak}_i : A \to A$*;*
- *for each* $i \in \mathbb{N}$*,* $\pi_i^A : A \to A$ *and* $\oplus_i^A : A \times A \to A$*;*
- $\rho : A \times \mathcal{A} \to A$*, where* $\mathcal{A}$ *is the set of boolean expressions.*

Intuitively, for each rule of the semantics, we have a corresponding operator that reflects the changes of the memory on the abstract domain. For each atomic statement s, the relation T_s characterizes the effect of the atomic semantic operation on the abstract domain. A particular instance of T that we usually consider is when s is a scalar assignment, i.e., $T_{i:=j}$, where $i \in \mathcal{N}_\mathcal{S}$ and $j \in \mathbb{Z}$. Note that we don't use the common *transfer functions* to define the abstract operators regarding atomic statements. Instead, we use relations, which encompasses the use of the more typical *backward* or *forward* functions. We can consider backward transfer functions by defining $a\ T_s\ b$ as $a = f_s(b)$ for a suitable f_s, and forward transfer functions by defining $a\ T_s\ b$ as $b = g_s(a)$ for a suitable g_s (an example of this is the safety analysis of Appendix 6.2.2). Also, atomic statements include Kernel and Scalar operations that can be arbitrarily complex and whose behavior can be better abstracted in a relation. For instance, in the case of verification, we will require that these atomic statements be specified with pre and postconditions that define the relation.

The operator $+_A$ abstracts the operator $+$ for memories (we omit the reference to the domain when it is clear from the context).

Given an $i \in \mathcal{N}_\mathcal{S}$ and $a \in A$, the function $\mathsf{weak}_i(a)$ removes any condition on the scalar variable i from a. It is used when processing a Forall task, with i being the iteration variable, to show that after execution, the value of the iteration variable is not relevant. To give more intuition about this operator, consider, for instance, that A is the lattice of first-order formulae (as is the case of the verification framework of Sect. 6.2.3), then $\mathsf{weak}_i(a)$ is defined as $\exists i.a$. If A has the form $\mathcal{N}_\mathcal{S} \to D$, where D is a lattice, then $\mathsf{weak}_i(a)$ can be defined as $a \sqcup \{i \to \top\}$, effectively removing any condition on i.

For each $i \in \mathbb{N}$, the operators $\{\pi_i^A\}_{i\in\mathbb{N}}$ and $\{\oplus_i^A\}_{i\in\mathbb{N}}$ abstract the operations π_i and $\oplus_i$ for memories (we omit the reference to the domain when it is clear from the context).

Finally, the function $\rho : A \times \mathcal{A} \to A$ is a transfer function used in an If task to update an abstract value depending on the test condition. It can be simply defined as $\rho(a, b) = a$, but this definition does not take advantage of knowing that b is true. If we have an expressive domain we can find a value that express this; for instance, in the lattice of logic formulae, we can define $\rho(a, b) = a \wedge b$.

To formalize the connection between the memory states and the abstract states, we assume a satisfaction relation $\models \subseteq \mathcal{H} \times A$ that is an approximation order, i.e., for all $\sigma \in \mathcal{H}$ and $a_1, a_2 \in A$, if $\sigma \models a_1$ and $a_1 \sqsubseteq a_2$ then $\sigma \models a_2$. The next definition formalizes the intuition given about the relation between the operators of an abstract interpretation and the semantics of programs. Basically, it states that satisfiability is preserved for each operator of the abstract interpretation. Note that we can also restate these lemmas and definitions in terms of Galois connections, since we can define a Galois connection from the relation $\models$ by defining $\gamma : A \to \mathcal{H}$ as $\gamma(a) = \{\sigma \in \mathcal{H} : \sigma \models a\}$.

$$\frac{\begin{array}{c}X \text{ the set of maximal elements of } H \text{ and } H' = H \setminus X\text{:}\\ \forall g \in X,\ \langle a\rangle \vdash g\,\langle a_g\rangle \qquad \langle \textstyle\sum_{g\in X}\ a_g\rangle \vdash \mathsf{Group}(H')\,\langle a'\rangle\end{array}}{\langle a\rangle \vdash \mathsf{Group}(H)\,\langle a'\rangle}\ [\mathbf{G}]$$

$$\frac{}{\langle a\rangle \vdash \mathsf{Group}(\emptyset)\,\langle a\rangle}\ [\mathbf{G}_\emptyset] \qquad \frac{s \in \mathsf{atomStmt} \quad a\ T_s\ a'}{\langle a\rangle \vdash s\,\langle a'\rangle}\ [\mathbf{A}]$$

$$\frac{\forall j,\ m \le j \le n \qquad a\ T_{i:=j}\ a_j \qquad \langle a_j\rangle \vdash G\,\langle a'_j\rangle}{\langle a\rangle \vdash \mathsf{Forall}\ i = m : n\ \mathsf{do}\ G\,\langle \textstyle\sum_{j=m}^{n} \mathsf{weak}_i(a'_j)\rangle}\ [\mathbf{F}]$$

$$\frac{\langle \rho(a, cond)\rangle \vdash G_1\,\langle a'\rangle \quad \langle \rho(a, \neg cond)\rangle \vdash G_2\,\langle a'\rangle}{\langle a\rangle \vdash \mathsf{If}\ cond\ \mathsf{then}\ G_1\ \mathsf{else}\ G_2\,\langle a'\rangle}\ [\mathbf{I}]$$

$$\frac{b \sqsubseteq a \quad \langle a\rangle \vdash G\,\langle a'\rangle \quad a' \sqsubseteq b'}{\langle b\rangle \vdash G\,\langle b'\rangle}\ [\mathbf{SS}] \qquad \frac{\langle \pi_i(a)\rangle \vdash G\,\langle a'\rangle}{\langle a\rangle \vdash \mathsf{Exec}_i(G)\,\langle a \oplus_i a'\rangle}\ [\mathbf{E}]$$

Fig. 6.2. Program analysis rules

Definition 6.5. *The abstract interpretation* $I = \langle A, T, +, \mathsf{weak}, \pi, \oplus, \rho \rangle$ *is consistent if the following holds for every* $\sigma, \sigma' \in \mathcal{H}$, $a, a_1, a_2 \in A$, $\mu, \mu_p \in \mathcal{M}$, $\tau \in \mathcal{T}$ *and* $cond \in \mathcal{A}$*:*

- *for every* $s \in \mathsf{atomStmt}$*, if* $\sigma \vdash s \rightarrow \sigma'$, $\sigma \models a$ *and* $a\ T_s\ a'$*, then* $\sigma' \models a'$*;*
- *if* $\sigma_1 \models a_1$ *and* $\sigma_2 \models a_2$ *then* $\sigma_1 +_\sigma \sigma_2 \models a_1 + a_2$*;*
- *if* $\mu_p, \langle \mu, \tau \rangle \models a$*, then for all* $k \in \mathbb{Z}$ $\mu_p, \langle \mu[i \mapsto k], \tau \rangle \models \mathsf{weak}_i(a)$*;*
- *if* $\sigma \models a$ *then* $\pi_i(\sigma) \models \pi_i(a)$*;*
- *if* $\sigma \models a$ *and* $\sigma' \models a'$*, then* $\sigma \oplus_i \sigma' \models a \oplus_i a'$*;*
- *if* $\sigma \models a$ *and* $\sigma \models_{\mathcal{A}} cond$*, then* $\sigma \models \rho(a, cond)$*.* [1]

Given an abstract interpretation I, a judgment is a tuple $\langle a \rangle \vdash_I G \langle a' \rangle$, where G is a program and $a, a' \in A$. We will omit the reference to I when it is clear from the context. A judgment is *valid* if it is the root of a derivation tree built using the rules in Fig. 6.2. The interpretation of a valid judgment $\langle a \rangle \vdash G \langle a' \rangle$ is that executing G in a memory that satisfies a, we end up in a memory satisfying a'. The following lemma claims that this is case, provided I is consistent.

Lemma 6.6 (Analysis Soundness). *Let* G *be a Sequoia program and assume that* $I = \langle A, T, +, \mathsf{weak}, \pi, \oplus, \rho \rangle$ *is a consistent abstract interpretation. For every* $a, a' \in A$ *and* $\sigma, \sigma' \in \mathcal{H}$*, if the judgment* $\langle a \rangle \vdash G \langle a' \rangle$ *is valid and* $\sigma \vdash G \rightarrow \sigma'$ *and* $\sigma \models a$ *then* $\sigma' \models a'$*.*

6.2.2 Program Safety

In this section we formalize a notion of program safety, i.e., a notion of parallel subtasks independence that ensures that written portions of the memory do not overlap. Then, when analyzing a safe program, one is discharged from considering every interleaving order in which its subtasks may be executed. The compiler for Sequoia described in [37] assumes that programs are safe, but does not check it. For our purposes of verification, checking this property is essential.

We assume every program G is provided with annotations R and W specifying respectively the regions of the memory that it may read and modify. The domains of the specification are the same for both R and W, indicating the set of scalar variables or array intervals that may be read or written:

$$R, W \in ({\mathcal{N}_{\mathcal{A}}}^+ \rightarrow \mathit{Interval}) \times \mathcal{P}({\mathcal{N}_{\mathcal{S}}}^+) \ ;$$

the domain *Interval* being defined as

$$\mathit{Interval} = \{(a, b) : a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\}_\perp \ .$$

[1] Given a memory σ and a boolean condition *cond*, the judgment $\sigma \models_{\mathcal{A}} cond$ states that the condition is valid in σ. The definition is standard so we omit it.

Given a region R, and memories $\sigma, \sigma' \in \mathcal{H}$, we write $\sigma \approx_R \sigma'$ if σ and σ' coincide in R. Given two memories $\sigma, \sigma' \in \mathcal{H}$, we denote $\mathit{Modified}(\sigma, \sigma')$ the set of extended locations that have different values in σ and σ'. We require the program annotations to be sound with respect to the program semantics. Let R, W be an annotation for program G. We say that R, W is sound with respect to the semantics of G if:

- for every $\sigma_1, \sigma_2 \in \mathcal{H}$ such that $\sigma_1 \approx_R \sigma_2$, if $\sigma_1 \vdash G \to \sigma_1'$ and $\sigma_2 \vdash G \to \sigma_2'$, then $\sigma_1' \approx_W \sigma_2'$.
- for every $\sigma, \sigma' \in \mathcal{H}$, if $\sigma \vdash G \to \sigma'$ then $\mathit{Modified}(\sigma, \sigma') \subseteq W$.

From the soundness of programs annotations it follows that the order of parallel program execution is irrelevant when tasks modify disjoint regions of the memory.

Lemma 6.7. *Assume programs G_1 and G_2, annotated respectively with the regions R_1,W_1 and R_2,W_2 s.t. $W_1 \cap (R_2 \cup W_2) = \emptyset$ and $W_2 \cap (R_1 \cup W_1) = \emptyset$. If $\sigma, \sigma_1, \sigma_2, \sigma_{12}, \sigma_{21} \in \mathcal{H}$ are memories s.t.*

$$\begin{array}{ll} \sigma \vdash G_1 \to \sigma_1 & \sigma_1 \vdash G_2 \to \sigma_{12} \\ \sigma \vdash G_2 \to \sigma_2 & \sigma_2 \vdash G_1 \to \sigma_{21}, \end{array}$$

then $\sigma_{12} = \sigma_{21}$.

Proof. Using soundness of I^R and I^W.

Using the region specification described above, we can determine whether a program is safe, using the safety judgment, $\vdash_{\text{Safe}} G$, defined by the rules shown in Fig. 6.3. The interesting rules are the rules for Group and Forall. We check that subtasks that can be executed in parallel have non-overlapping regions. More specifically, the writing region of one task cannot overlap with neither the reading nor the writing region of another independent task. This ensures that the final memory does not depend on the order in which tasks are executed. In the case of Forall, there is an overlap between the variables written by each subtask, namely the iteration variable (since this variable is written at the beginning of the execution). We allow this overlap, because we consider it as a local variable, that is not to be used after executing the Forall, since its value is not defined. Hence, the final memory does not depend on the order of execution of parallel subtasks.

We conclude this section with a brief discussion of the difference between our semantics (defined in Fig. 6.1) and the original semantics for Sequoia defined in [37]. The difference lies in the rule for Group(H): whereas we require that X is the complete set of maximal elements, and is thus uniquely determined from the program syntax, the original semantics does not require X to be the complete set of maximal elements, but only a subset of it. This means that, in the original semantics, the order of execution of parallel subtasks is not deterministic, and therefore, our semantics is a restriction of the original one. However, Proposition 6.8 justifies the change we made to the

$$\frac{G \in \mathsf{atomStmt}}{\vdash_{\text{Safe}} G} \qquad \frac{\pi_i(s) \vdash_{\text{Safe}} G}{s \vdash_{\text{Safe}} \mathsf{Exec}_i(G)} \qquad \frac{s \vdash_{\text{Safe}} G_1 \qquad s \vdash_{\text{Safe}} G_2}{s \vdash_{\text{Safe}} \mathsf{If}\ b\ \mathsf{then}\ G_1\ \mathsf{else}\ G_2}$$

$$\frac{\forall G, G' \in H \text{ with } G \text{ and } G' \text{ unrelated in } H \qquad R,W \text{ sound annotation for } G \qquad R',W' \text{ sound annotation for } G' \qquad (R \cup W) \cap W' = \emptyset \qquad (R' \cup W') \cap W = \emptyset}{\vdash_{\text{Safe}} \mathsf{Group}(H)}$$

$$\frac{\forall k,\ R_k,W_k \text{ sound annotation for } G \qquad \forall k, k',\ (R_k \cup W_k) \cap W_{k'} = \{j\}}{\vdash_{\text{Safe}} \mathsf{Forall}\ j = s : e\ \mathsf{do}\ G}$$

Fig. 6.3. Program Safety

Sequoia semantics, since they are equivalent in the sense that our semantics can simulate any run of the unrestricted semantics—provided the program is safe, i.e., parallel subtasks are independent, as intended in Sequoia[2]. The following proposition formalizes the relation between the original semantics and the semantics defined in Fig. 6.1. To differentiate the semantics, we use $\rightarrow_O$ to denote the original semantics relation.

Proposition 6.8. *Assume a program G s.t. $\vdash_{\text{Safe}} G$, and a memory hierarchy $\sigma \in \mathcal{H}$. If $\sigma \vdash G \rightarrow \sigma_1$ and $\sigma \vdash G \rightarrow_O \sigma_2$, then $\sigma_1 = \sigma_2$.*

Proof (Proposition 6.8). We want to prove that the order of execution of independent subtasks does not matter for safe programs. While the original semantics seems to be non-deterministic because of the choice we can make in the Group *rule, for safe programs, this choice does not matter as the final result will always be the same.*

We consider yet another rule for Group*, where we execute all subtasks in sequential order (we use the symbol $\rightarrow_1$ to differentiate from other semantics):*

$$\frac{\{g_1, g_2, \ldots, g_n\} = H \qquad \sigma = \sigma_1 \qquad \forall i \in \{1, \ldots, n\},\ \sigma_i \vdash g_i \rightarrow_1 \sigma_{i+1}}{\sigma \vdash \mathsf{Group}(H) \rightarrow_1 \sigma_{n+1}}$$

The condition in the previous rule is that the order $g_1, \ldots, g_n$ respects the dependencies of the graph H. Note that this order is not unique.

Given a safe program G s.t. $\sigma \vdash G \rightarrow \sigma'$ we can show by induction on the semantics that $\sigma \vdash G \rightarrow_1 \sigma'$. In fact, we can show, using the previous lemma, that the order in which we execute the subtasks of a Group *does not affect the final memory.*

Using a similar reasoning, we can show that if $\sigma \vdash G \rightarrow_O \sigma''$, then $\sigma \vdash G \rightarrow_1 \sigma''$. Combining both results, we infer that $\sigma' = \sigma''$.

[2] This notion of safety is similar to the notion of strict and-parallelism of logic programming [34].

6.2.3 Program Verification

We now define a verification framework $I = \langle \mathcal{C}, T, +_{\mathcal{C}}, \mathsf{weak}, \pi, \oplus, \rho \rangle$ as an instance of the abstract interpretation, where $\mathcal{C}$ is the prelattice of first-order formulae. Before defining I, we need some preliminary definitions.

The extended set of scalar names, ${\mathcal{N}_{\mathcal{S}}}^+$, is defined as

$${\mathcal{N}_{\mathcal{S}}}^+ = \mathcal{N}_{\mathcal{S}} \cup \{x^{\uparrow} : x \in \mathcal{N}_{\mathcal{S}}\} \cup \{x^{\downarrow^{i_1} \ldots \downarrow^{i_k}} : x \in \mathcal{N}_{\mathcal{S}} \wedge k \in \mathbb{N} \wedge i_1, \ldots, i_k \in \mathbb{N}\} \ .$$

We define, in a similar way, the sets ${\mathcal{N}_{\mathcal{A}}}^+$, $\mathcal{N}^+$, and $\mathcal{L}^+$ of extended locations. These sets allow us to refer to variables at all levels of a memory hierarchy, as is shown by the following definition. Given $\sigma \in \mathcal{H}$, with $\sigma = \mu_p, \langle \mu, \tau \rangle$, and $l \in \mathcal{L}^+$, we define $\sigma(l)$ with the following rules:

$$\sigma(l) = \begin{cases} \mu_p(x) & \text{if } l = x^{\uparrow} \\ \mu(x) & \text{if } l = x \\ (\mu, \tau_{i_1})(x^{\downarrow^{i_2} \ldots \downarrow^{i_k}}) & \text{if } l = x^{\downarrow^{i_1} \downarrow^{i_2} \ldots \downarrow^{i_k}} \end{cases} .$$

We also define the functions $\uparrow^i, \downarrow^i : {\mathcal{N}_{\mathcal{S}}}^+ \to {\mathcal{N}_{\mathcal{S}}}^+$ with the following rules:

$$\begin{array}{ll} \downarrow^i(x) = x^{\downarrow^i} & \uparrow^i(x) = x^{\uparrow} \\ \downarrow^i(x^{\downarrow^{j_1} \ldots \downarrow^{j_n}}) = x^{\downarrow^i \downarrow^{j_1} \ldots \downarrow^{j_n}} & \uparrow^i(x^{\downarrow^i \downarrow^{j_1} \ldots \downarrow^{j_k}}) = x^{\downarrow^{j_1} \ldots \downarrow^{j_k}} \\ \downarrow^i(x^{\uparrow}) = x \ . & \end{array}$$

Note that $\downarrow^i$ is a total function, while $\uparrow^i$ is undefined in $x^{\uparrow}$ and $x^{\downarrow^j \downarrow^{j_1} \ldots \downarrow^{j_k}}$ if $j \neq i$. These functions are defined likewise for ${\mathcal{N}_{\mathcal{A}}}^+$, $\mathcal{N}^+$, and $\mathcal{L}^+$.

Given a formula ϕ, we obtain $\downarrow^i \phi$ by substituting every free variable $v \in \mathcal{N}^+$ of ϕ with $\downarrow^i v$. In the same way, the formula $\uparrow^i \phi$ is obtained by substituting every free variable $v \in \mathcal{N}^+$ of ϕ by $\uparrow^i v$; if $\uparrow^i v$ is not defined, we substitute v by a fresh variable, and quantify existentially over all the introduced fresh variables.

Definition of +.

To define the operator + we require that each subprogram comes annotated with the sets SW and AW specifying, respectively, the scalar variables and the array ranges that it may modify. Given two programs G_1 and G_2 annotated, respectively, with the modifiable regions SW_1, AW_1 and SW_2, AW_2, and the postconditions Q_1 and Q_2, we define $Q_1 + Q_2$ as $Q_1' \wedge Q_2'$, where Q_1' is the result of existentially quantifying in Q_1 the variables that may be modified by G_2. More precisely, $Q_1' = \exists X'.\ Q_1[{}^{X'}/_{X}] \wedge \bigwedge_{A[m,n] \in AW_1} A'[m,n] = A[m,n]$, X representing the set of scalar and array variables in $SW_2 \cup AW_2$ and X' a set of fresh variables (and similarly with Q_2).

To explain the intuition behind this definition, assume two tasks G_1 and G_2 that execute in parallel with postconditions Q_1 and Q_2. After verifying

that each G_i satisfies the postcondition Q_i, one may be tempted to conclude that after executing both tasks, the resulting state satisfies $Q_1 \wedge Q_2$. The reason for which we do not define $Q_1 + Q_2$ simply as $Q_1 \wedge Q_2$ is that while Q_1 may be true after executing G_1, Q_1 may state conditions over variables that are not modified by G_1 but are modified by G_2. Then, since from the definition of the operator $+$ in the semantic domain the value of a variable not modified by G_1 is overwritten with a new value if modified by G_2, Q_1 may be false in the final memory state after executing G_1 and G_2 in parallel.

For the definition of $Q_1 + Q_2$ to be sound we require the annotations SW_1, AW_1 and SW_2, AW_2 to be correct. For this, we can use a static analysis or generate additional proof obligations to validate the program annotations. However, for space constraints and since such analysis can be applied earlier and independently of the verification framework, we do not consider this issue.

We generalize the operator $+$ for a set of postconditions $\{\phi_i\}_{i \in I}$ and a set of specifications of modified variables $\{SW_i\}_{i \in I}$ and $\{AW_i\}_{i \in I}$, by defining $\sum_{i \in I} \phi_i$ as $\bigwedge_{i \in I} \phi'_i$ where $\phi'_i = \exists X'.\ \phi_i[{}^{X'}/_{X}] \wedge \bigwedge_{A[m,n] \in AW_i} A[m,n] = A'[m,n]$, s.t. X represents every scalar or array variable in $\{SW_j\}_{j \neq i \in I}$ or $\{AW_j\}_{j \neq i \in I}$, and X' a set of fresh variables. If an assertion ϕ_i refers only to scalar and array variables that are not declared as modifiable by other member $j \neq i$, we have $\phi'_i \Rightarrow \phi_i$.

Definition of the weakest precondition.

The weakest precondition for a copy operation $\mathsf{Copy}^{\uparrow}(A[m,n], B[k])$ to the parent memory is defined w.r.t. postcondition Q as $Q[{}^{B^{\uparrow} \oplus [k,k+n-m] \mapsto A[m]}/_{B^{\uparrow}}]$. Given two arrays A and B and scalars m, n and k, we write $B \oplus [m,n] \mapsto A[k]$ to mean the array obtained from B by replacing the range $[m,n]$ with values from $A[k, k+n-m]$. The weakest precondition for other types of copy (from the parent memory, and intra-memory) are treated similarly.

The case of Kernel and Scalar are similar, so we will only consider the former. For each function appearing in a Kernel or Scalar statement, we assume that we have a pre and a postcondition. To illustrate, let us consider a function f with one parameter of array-type, and also returning and array. The pre and postcondition of f are first order formulae. Also, the precondition can refer to the input variable (with arg), to the range of indices read of the argument (with $\mathsf{arg}_{\mathsf{start}}$ and $\mathsf{arg}_{\mathsf{end}}$), and to the range of indices written of the result array (with $\mathsf{res}_{\mathsf{start}}$ and $\mathsf{res}_{\mathsf{end}}$). The postcondition can also refer to the result variable (with res).

Lets assume that f doubles each value of the input array, so the pre and postcondition could be written as follows:

$$\text{Pre} = (\mathsf{res}_{\mathsf{end}} - \mathsf{res}_{\mathsf{start}} = \mathsf{arg}_{\mathsf{end}} - \mathsf{arg}_{\mathsf{start}})$$
$$\text{Post} = \forall x,\ \mathsf{res}_{\mathsf{start}} \leq x \leq \mathsf{res}_{\mathsf{end}} \Rightarrow \mathsf{res}[x] = 2 \cdot \mathsf{arg}[x - \mathsf{res}_{\mathsf{start}} + \mathsf{arg}_{\mathsf{start}}]$$

The precondition states that the input and output array must have the same length, and the postcondition states that all values of the result array are the double of the values of the argument, for the corresponding ranges.

The weakest precondition for a function f with precondition φ and postcondition ψ is the following:

$$\{P\} \vdash \mathsf{Kernel}\langle A[k,l] = f(B[m,n])\rangle \ \{Q\}\,,$$

where

$$\begin{aligned} P &= P' \wedge \forall \mathsf{res},\ Q' \Rightarrow Q[{}^{\mathsf{res}}/_{A}] \\ P' &= \varphi[{}^{B}/_{\mathsf{arg}}][{}^{m}/_{\mathsf{arg}_{\mathsf{start}}}][{}^{n}/_{\mathsf{arg}_{\mathsf{end}}}][{}^{k}/_{\mathsf{res}_{\mathsf{start}}}][{}^{l}/_{\mathsf{res}_{\mathsf{end}}}] \\ Q' &= \psi[{}^{B}/_{\mathsf{arg}}][{}^{m}/_{\mathsf{arg}_{\mathsf{start}}}][{}^{n}/_{\mathsf{arg}_{\mathsf{end}}}][{}^{k}/_{\mathsf{res}_{\mathsf{start}}}][{}^{l}/_{\mathsf{res}_{\mathsf{end}}}] \end{aligned}$$

Definition of other components of I.

They are defined as follows:

- for each $s \in \mathsf{atomStmt}$, the relation T_s is defined from the weakest precondition transformer wp_s, as $\phi\ T_s\ \phi'$ if $\phi \Rightarrow \mathsf{wp}_s(\phi')$ for every logic formulae ϕ and ϕ'. For Kernel and Scalar statements, we assume that we have a pre and postcondition specifying their behavior;
- $\mathsf{weak}_i(\phi) = \exists i,\ \phi$, where $i \in {\mathcal{N}_{\mathcal{S}}}^+$;
- $\pi_i(\phi) = \uparrow^i \phi$, where $i \in \mathbb{N}$;
- $\phi_1 \oplus_i \phi_2 = \overline{\phi_1}^{\,i} \wedge \downarrow^i \phi_2$, where $i \in \mathbb{N}$, and $\overline{\phi_1}^{\,i}$ is obtained from ϕ_1 by replacing every variable of the form x or $x^{\downarrow^i \downarrow^{j_1} \cdots \downarrow^{j_k}}$ with a fresh variable and then quantifying existentially all the introduced fresh variables;
- $\rho(\phi, cond) = \phi \wedge cond$.

The satisfaction relation $\sigma \models \phi$ is defined as the validity of $[\![\phi]\!]\sigma$, the interpretation of the formula ϕ in the memory state σ. To appropriately adapt a standard semantics $[\![.]\!]$ to a hierarchy of memories, it suffices to extend the interpretation for the extended set of variables $\mathcal{N}^+$, as $[\![n]\!]\sigma = \sigma(n)$ for $n \in \mathcal{N}^+$.

In the rest of the section, we denote as $\{P\} \vdash G\ \{Q\}$ the judgments in the domain of logical formulae, and P and Q are said to be pre and postconditions of G respectively. If the judgment $\{P\} \vdash G\ \{Q\}$ is valid, and the program starts in a memory σ that satisfies P and finishes in a memory σ', then σ' satisfies Q. The proposition below formalizes this result.

Proposition 6.9 (Verification Soundness). *Assume that $\{P\} \vdash G\ \{Q\}$ is a valid judgment and that $\sigma \vdash G \rightarrow \sigma'$, where G is a program, P, Q are assertions, and $\sigma, \sigma' \in \mathcal{H}$. If $\sigma \models P$ then $\sigma' \models Q$.*

6.2.4 Example Program

We illustrate the verification with an example. Consider a program, G_{Add}, that add two input arrays (A and B) producing on output array C. The code of the program is given by the following definitions:

$G_{\text{Add}} := \mathsf{Exec}_0(\mathsf{Forall}\ i = 0 : n-1\ \mathsf{do}\ \text{Add})$
$\text{Add} := \mathsf{Group}((\text{CopyAX} \parallel \text{CopyBY}); \text{AddP}; \text{CopyZC})$
$\text{CopyAX} := \mathsf{Copy}^{\downarrow}(A[i.S, (i+1)S], X[i.S, (i+1)S])$
$\text{CopyBY} := \mathsf{Copy}^{\downarrow}(B[i.S, (i+1)S], Y[i.S, (i+1)S])$
$\text{AddP} := \mathsf{Kernel}\langle Z[i.S, (i+1)S] = \text{VectAdd}(X[i.S, (i+1)S], Y[i.S, (i+1)S])\rangle$
$\text{CopyZC} := \mathsf{Copy}^{\uparrow}(Z[i.S, (i+1)S], C[i.S, (i+1)S])$

Assume that the arrays have size $n.S$, and note that the program is divided in n parallel subtasks, each operating on different array fragments, of size S. The best value for S may depend on the underlying architecture.

It is easy to see that this program is safe, since each subtask writes on a different fragment of the arrays.

We show, using the verification framework, how to derive the judgment $\{\mathsf{true}\} \vdash G_{\text{Add}}\ \{\text{Post}\}$, where $\text{Post} = \forall k,\ 0 \leq k < n.S \Rightarrow C[k] = A[k] + B[k]$. Using the rules **[A]**, **[G]** and **[SS]** we derive, for each $j \in [0 \ldots n-1]$, the following:

$$\{\mathsf{true}\} \vdash \text{Add}\ \{Q_i\}\,, \tag{6.1}$$

where $Q_i = \forall k,\ i.S \leq k < (i+1)S \Rightarrow C^{\uparrow}[k] = A^{\uparrow}[k] + B^{\uparrow}[k]$. Applying the rule **[F]** on (6.1) we obtain

$$\{\mathsf{true}\} \vdash \mathsf{Forall}\ i = 0 : n-1\ \mathsf{do}\ \text{Add}\ \left\{\textstyle\sum_{0 \leq j < n} Q_j\right\}\ . \tag{6.2}$$

It is not difficult to see that $\sum_{0 \leq j < n} Q_j \Rightarrow \bigwedge_{0 \leq j < n} Q_j$, since each Q_j does not refer to variables that are modified by Add, for $i \neq j$. Applying the subsumption rule to (6.2), we obtain

$$\{\mathsf{true}\} \vdash \mathsf{Forall}\ i = 0 : n-1\ \mathsf{do}\ \text{Add}\ \{Q\} \tag{6.3}$$

where $Q = \forall k,\ 0 \leq k < n.S \Rightarrow C^{\uparrow}[k] = A^{\uparrow}[k] + B^{\uparrow}[k]$. Finally, applying rule **[E]** to (6.3), we obtain the desired result, since $\text{Post} = \downarrow^0 Q$.

6.3 Certificate translation

In this section, we focus on the interplay between program optimization and program verification. To maximize the performance of applications, the Sequoia compiler performs program optimizations such as code hoisting, instruction scheduling, and SPMD distribution. We show, for common optimizations

described in [37], that program optimizations transform provably correct programs into provably correct programs. More precisely, we provide an algorithm to transform a derivation for the original program into a derivation for the transformed program. The problem of transforming provably correct programs into provably correct programs is motivated by research in Proof Carrying Code (PCC) [50, 49], and in particular by our earlier work on certificate translation [7, 10].

We start by extending the analysis setting described in previous sections with a notion of certificates, to make it suitable for a PCC architecture. Then, we describe certificate translation in the presence of three optimizations: SPMD distribution, Exec Grouping, and Copy grouping.

Certified setting.

In a PCC setting, a program is distributed with a checkable certificate that the code complies with the specified policy. To extend the verification framework defined in Section 6.2.3 with a certificate infrastructure, we capture the notion of checkable proof with an abstract proof algebra.

Definition 6.10 (Certificate infrastructure). *A certificate infrastructure consists on a proof algebra $\mathcal{C}$ that assigns to every $\phi \in \mathcal{C}$ a set of certificates $\mathcal{C}(\vdash \phi)$. We assume that $\mathcal{C}$ is sound, i.e., for every $\phi \in \mathcal{C}$, if ϕ is not valid, then $\mathcal{C}(\phi) = \emptyset$. In the sequel, we write $c :\vdash \phi$ instead of $c \in \mathcal{C}(\phi)$.*

As in previous chapters, we do not commit to an specific representation of certificates. However, to give an intuition, we can define them, for example, in terms of the Curry-Howard isomorphism by considering $\mathcal{C}(\phi) = \{e \in \mathcal{E} \mid \langle\rangle \vdash e : \phi\}$, where $\mathcal{E}$ is the set of expressions and $\vdash e : \phi$ a typing judgment in some λ-calculus.

The operations of the proof algebra are standard, to the exception of the operator subst that allows to substitute selected instances of equals by equals, and of the operator *ring*, which establishes ring equalities that will be used to justify the optimizations.

In addition, we refine the notion of certified analysis judgment, to enable code consumers to check whether a judgment is a valid judgment. To this end, the rule definitions are extended to incorporate certificates attesting the validity of the required (a priori undecidable) logical formulae.

Definition 6.11 (Certified Verification Judgment). *We say that the verification judgment $\{\Phi\}\vdash G\ \{\Psi\}$ is* certified *if it is the root of a derivation tree, built from the rules in Fig. 6.2, such that every application of the subsumption rule*

$$\frac{\phi \Rightarrow \phi' \quad \{\phi'\}\vdash G\ \{\psi'\} \quad \psi' \Rightarrow \psi}{\{\phi\}\vdash G\ \{\psi\}}\textbf{[SS]}$$

is accompanied with certificates c and c' s.t. $\mathsf{c} :\vdash \phi \Rightarrow \phi'$ and $\mathsf{c}' :\vdash \psi' \Rightarrow \psi$. Furthermore, every application of the relation $\phi\ T_s\ \phi'$ is accompanied with a certificate $\mathsf{c} :\vdash \phi \Rightarrow \mathsf{wp}_s(\phi')$.

$$
\begin{array}{ll}
\mathsf{intro}_{\mathsf{true}} & : \mathcal{C}(\Gamma \vdash \mathsf{true}) \\
\mathsf{axiom} & : \mathcal{C}(\Gamma; A; \Delta \vdash A) \\
\mathsf{ring} & : \mathcal{C}(\Gamma \vdash n_1 = n_2) \quad \text{if } n_1 = n_2 \text{ is a ring equality} \\
\mathsf{intro}_{\wedge} & : \mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma \vdash B) \to \mathcal{C}(\Gamma \vdash A \wedge B) \\
\mathsf{elim}^{\mathsf{l}}_{\wedge} & : \mathcal{C}(\Gamma \vdash A \wedge B) \to \mathcal{C}(\Gamma \vdash A) \\
\mathsf{elim}^{\mathsf{r}}_{\wedge} & : \mathcal{C}(\Gamma \vdash A \wedge B) \to \mathcal{C}(\Gamma \vdash B) \\
\mathsf{intro}_{\Rightarrow} & : \mathcal{C}(\Gamma; A \vdash B) \to \mathcal{C}(\Gamma \vdash A \Rightarrow B) \\
\mathsf{elim}_{\Rightarrow} & : \mathcal{C}(\Gamma \vdash A \Rightarrow B) \to \mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma \vdash B) \\
\mathsf{elim}_{=} & : \mathcal{C}(\Gamma \vdash e_1 = e_2) \to \mathcal{C}(\Gamma \vdash A[{}^{e_1}\!/_r]) \to \mathcal{C}(\Gamma \vdash A[{}^{e_2}\!/_r]) \\
\mathsf{subst} & : \mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma[{}^{e}\!/_r] \vdash A[{}^{e}\!/_r]) \\
\mathsf{weak}_{\sqcap} & : \mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma; \Delta \vdash A) \\
\mathsf{intro}_{\forall} & : \mathcal{C}(\Gamma \vdash A) \to \mathcal{C}(\Gamma \vdash \forall r.A) \qquad \text{if } r \text{ is not in } \Gamma \\
\mathsf{elim}_{\forall} & : \mathcal{C}(\Gamma \vdash \forall r.A) \to \mathcal{C}(\Gamma \vdash A)
\end{array}
$$

Fig. 6.4. Proof Algebra (excerpt)

6.4 General framework

In this section, we consider the translation of certified verification judgments when a program G' is derived from the original program G by a structure preserving transformation. To this end, we require that the analysis that motivates the transformation ensures that the semantics is preserved. The latter condition is formulated in terms of the underlying verification framework as explained later in this section. This category of transformations covers several optimizations that improve the efficiency of an atomic sub-program exploiting conditions ensured by previously executed statements, including standard optimizations such as constant propagation or common sub-expression elimination (which are not treated here, but refer to [7, 10]). We will illustrate the transformation with copy propagation.

6.4.1 Certifying analyzers

Certificate translation along a program transformation may require that the analysis that justifies the transformation is certified. That is, that the result of the analysis can be specified in the domain of a logical formulae, and that the validity of this specification can be ensured by the existence of an automatically generated certificate.

In this section, we provide sufficient conditions to augment analyzers to automatically generate certificates for the result of the analysis. In the following, these extended analyzers are called certifying analyzers.

Consider an abstract interpretation $I = \langle A, f, T, +, \mathsf{weak}, \pi, \oplus, \rho\rangle$ representing a static analysis, performed before the program transformation, and assume that $\langle a\rangle \vdash G\, \langle a'\rangle$ is a valid analysis judgment. To formalize the representation of elements of the abstract domain A, we assume that A is related to the domain of logic formulae by a concretization function $\gamma : A \to \mathcal{C}$. That is, a function that for any $a \in A$, returns an interpretation of a as a logic formula.

We provide sufficient conditions to generate a certified verification judgment $\{\gamma(a)\} \vdash G\ \{\gamma(a')\}$ from the valid analysis judgment $\langle a\rangle \vdash G\, \langle a'\rangle$.

Definition 6.12. *An abstract interpretation $I = \langle A, f, T, +, \mathsf{weak}, \pi, \oplus, \rho\rangle$ is consistent with the verification framework if we have*

- *for every $a_1, a_2 \in A$ s.t. $a_1 \sqsubseteq a_2$, a certificate $\mathsf{monot}_\gamma : \gamma(a_1) \Rightarrow \gamma(a_2)$*
- *for every $a_1, a_2 \in D, s \in \mathsf{atomStmt}$ s.t. $a_1\ T_s\ a_2$, a certificate $\mathsf{cons}_s(a) : \gamma(a_1) \Rightarrow \mathsf{wp}_s(\gamma(a_2))$;*
- *for every $a, a' \in A$, the certificates $\mathsf{distrib}_{+,\gamma} :\vdash \gamma(a) + \gamma(a') \Rightarrow \gamma(a + a')$ and $\mathsf{distrib}_{\sqcap,\gamma} :\vdash \gamma(a \sqcap a') \Rightarrow \gamma(a) \wedge \gamma(a')$;*
- *for every $a \in A$, $i \in \mathcal{N}_\mathcal{S}$, $\mathsf{c}_{\mathsf{weak}} :\vdash \mathsf{weak}_i(\gamma(a)) \Rightarrow \gamma(\mathsf{weak}_i(a))$;*
- *for every $i \in \mathbb{Z}$ $a, a' \in A$, the certificates $\mathsf{c}_{\pi_i} :\vdash \pi_i(\gamma(a)) \Rightarrow \gamma(\pi_i(a))$ and $\mathsf{c}_{\oplus_i} :\vdash \gamma(a) \oplus_i \gamma(a') \Rightarrow \gamma(a \oplus_i a')$; and*
- *for every $a \in A$ and $b \in \mathcal{A}$, a certificate $\mathsf{c}_\rho : \gamma(a) \wedge b \Rightarrow \gamma(\rho(a, b))$.*

The following result states that a valid analysis judgment that motivates a program transformation is certifiable, as long as the analysis I is consistent with the verification framework.

Lemma 6.13. *Let $\langle a\rangle \vdash G\, \langle a'\rangle$ be a valid analysis judgment. Then, provided I is consistent with the verification framework, $\{\gamma(a)\} \vdash G\ \{\gamma(a')\}$ is a certified verification judgment*

Proof. The proof is by induction on the derivation of $\langle a\rangle \vdash G\, \langle a'\rangle$. We consider only some representative cases.

- *Base case. Last rule applied is* **[A]**. *Then $a\ T_s\ a'$, with $s = G$. By consistency of the analysis I, we have a certificate $\mathsf{cons} :\vdash \gamma(a) \Rightarrow \mathsf{wp}_s(\gamma(a'))$ Then we have a certified application of the rule* **[A]** *in the domain of the verification environment, to get the judgment $\{\gamma(a)\} \vdash s\ \{\gamma(a')\}$.*
- *Last rule applied is* **[SS]**. *Then we have a sub-derivation tree for the judgment $\langle b\rangle \vdash G\, \langle b'\rangle$ and $a \sqsubseteq b$ and $b' \sqsubseteq a'$. We know, by I.H., that the judgment $\{\gamma(b)\} \vdash G\ \{\gamma(b')\}$ is certified. By monotonicity of γ we have certificates for $\gamma(a) \Rightarrow \gamma(b)$ and $\gamma(b') \Rightarrow \gamma(a')$ and then, by subsumption, a certified judgment $\{\gamma(a)\} \vdash G\ \{\gamma(a')\}$*
- *Last rule applied is* **[F]**. *Then G has the form $\mathsf{Forall}\ i = m : n\ \mathsf{do}\ G'$, a is such that $aT_{i:=j}a_j$ for every $j \in [m, n]$ and a' is equal to $\sum_{j=m}^{n} \mathsf{weak}_i(a'_j)$, where for every $j \in [m, n]$ there is a sub-derivation tree for the judgment $\langle a_j\rangle \vdash G'\, \langle a'_j\rangle$. By I.H., we know the judgments $\{\gamma(a_j)\} \vdash G'\ \{\gamma(a'_j)\}$ are certified. From the existence of cons we have a certificate for $\gamma(a) \Rightarrow$*

$\mathsf{wp}_{i:=j}(\gamma(a_j))$, *to certify the relation* $\gamma(a)T_{i=j}\gamma(a_j)$ *in the domain of the verification environment, for every* $j \in [m, n]$.
To apply a certified subsumption rule, it remains to show that we have a certificate for $\sum_{j=m}^{n} \mathsf{weak}_i(\gamma(a'_j)) \Rightarrow \gamma(\sum_{j=m}^{n} \mathsf{weak}_i(a'_j))$. *The existence of this certificate follows from the application of* $\mathsf{distrib}_{+,\gamma}$ *and* $\mathsf{c}_{\mathsf{weak}}$.

- *Last rule applied is* **[E]**. *We know that* G *is of the form* $\mathsf{Exec}_i(G')$, a' *is equal to* $a \oplus_i a''$ *for some* $a'' \in A$, *and that there sub-derivation tree for the judgment* $\langle \pi_i(a) \rangle \vdash G' \langle a'' \rangle$. *Then, by I.H., the judgment* $\langle \gamma(\pi_i(a)) \rangle \vdash G' \langle \gamma(a'') \rangle$ *is certified. By application of* c_{π_i}, *we get a certificate for* $\pi_i(\gamma(a)) \Rightarrow \gamma(\pi_i(a))$ *and then, by subsumption, a certified judgment* $\langle \pi_i(\gamma(a)) \rangle \vdash G' \langle \gamma(a'') \rangle$. *Applying the rule* **[E]**, *we get* $\langle \gamma(a) \rangle \vdash G' \langle \gamma(a) \oplus_i \gamma(a'') \rangle$. *Finally, from* $\mathsf{c}_{\oplus_i}$ *and application of the subsumption rule* **[CSS]** *we have the certified judgment* $\langle \gamma(a) \rangle \vdash G' \langle \gamma(a \oplus_i a'') \rangle$.

The existence of certifying analyzers is central to certificate translation, since given an original verification judgment, we often need to refine its derivation by strengthening the annotations with the result of the analysis.

6.4.2 Certificate translation.

Let G' be a program derived from G by a structure preserving transformation, i.e., G and G' have the same structure but differ on the leaves of the abstract syntax tree.

Definition 6.14 (Justified transformation). *Let* s *and* s' *be atomic statements and* R *a logic formula. The substitution of* s *by* s' *is justified by* R *if for every assertion* ϕ, *we have a certificate* $\mathsf{justif} :\vdash R \wedge \mathsf{wp}_s(\phi) \Rightarrow \mathsf{wp}_{s'}(\phi)$. *We say that a derivation tree for the judgment* $\{P\} \vdash G \{Q\}$ *justifies a structure preserving transformation from* G *to* G', *if the substitution of every atomic subprogram* g *in* G *by* g' *is justified by a precondition* R *s.t.* $\{R\} \vdash g \{R'\}$ *is the corresponding derivation sub-tree of* $\{P\} \vdash G \{Q\}$.

The following result, in combination with Lemma 6.13, states that certificate translation from G to G' is feasible if G' is derived from G by a structure preserving transformation that is justified by a valid analysis judgment.

Lemma 6.15. *Let* G' *by a program derived from* G *by a structure preserving transformation. Assume that the transformation is justified by the certified judgment* $\{R\} \vdash G \{R'\}$, *and consider an original certified judgment* $\{P\} \vdash G \{Q\}$. *Then, we can build a derivation tree for the judgment* $\{P \wedge R\} \vdash G' \{Q \wedge R'\}$ *for the transformed program.*

In the following paragraphs we consider a common Sequoia optimization [37] as an instance of the general characterization above.

6.4.3 Copy propagation for arrays

In sequential programming languages, copy propagation is an optimization that replaces the occurrence of a variable by another variable if they are known to contain the same value. In that case, the transformation is intended to reduce the live range of the replaced variable, cleaning the effect of other optimizations and enabling further transformations.

Since array operations are frequent in programs targeting data intensive applications, it is of interest to extend traditional copy propagation to consider copy operations between arrays. In this case, removing copy operations is motivated mainly by the reduction of the bulk data transfer involved in each copy operation.

Naturally, this transformation requires a richer analysis domain than that of classical copy propagation, since we need to relate not simply arrays but array regions.

Consider an abstract interpretation $I = \langle A, \downarrow, +, \mathsf{weak}, \pi, \oplus, \rho \rangle$ with domain $A = \mathcal{P}(\mathcal{N}_\mathcal{A} \times \mathit{Interval} \times \mathcal{N}_\mathcal{A} \times \mathit{Interval})$, where

$$\mathit{Interval} = \{(a, b) : a \in \mathbb{Z} \cup \{-\infty\}, b \in \mathbb{Z} \cup \{\infty\}, a \leq b\}_\perp,$$

is the lattice for interval analysis. An element $(A, [m, n], A', [m', n'])$ in $\mathcal{N}_\mathcal{A} \times \mathit{Interval} \times \mathcal{N}_\mathcal{A} \times \mathit{Interval}$, denoted $A[m, n] = A'[m', n']$ for readability, is satisfied by a memory hierarchy $\sigma \in \mathcal{H}$ if the intervals $[m, n]$ and $[m', n']$ are equally sized and the arrays A and A' coincide on that ranges. An element a of the abstract domain A is satisfied by a memory hierarchy $\sigma \in \mathcal{H}$ if for all $A[m, n] \in S$, σ satisfies $A[m, n]$.

Definition 6.16. *Let a be an abstract element such that $A[m, n] = A'[m', n'] \in a$. Consider an atomic statement s that reads an array range $A[x, y]$ such that $[x, y] \subseteq [m, n]$. Then, the judgment $\langle a \rangle \vdash s \langle a' \rangle$ induces a transformation of s into s' if s' is the result of substituting every access of $A[i]$ in s by $A'[i-m+m']$.*

For instance, let s be the statement $\mathsf{Kernel}\langle Z = VectAdd(B[0, k], C[0, k])\rangle$ and $a \in A$ such that $B[0, k] = A[m, m + k] \in a$. Then, for any $a' \in A$ the judgment $\langle a \rangle \vdash s \langle a' \rangle$ induces the substitution of the atomic statement s by the statement $\mathsf{Kernel}\langle Z = VectAdd(A[m, m + k], C[0, k])\rangle$.

Consider the certified judgment $\{\Phi\} \vdash G \{\Psi\}$. A first requirement to translate the judgment along array copy propagation is a certificate of the analysis judgment $\langle a \rangle \vdash G \langle a' \rangle$. To this end, we interpret the result of the analysis with a concretization function $\gamma : A \to \mathcal{C}$ defined as

$$\gamma(A[m, n] = A'[m', n']) \doteq (n - m = n' - m') \wedge (\forall_{m \leq i \leq n}.\ A[i] = A'[i + m' - m])$$

Lemma 6.17. *Consider a program G and a valid analysis judgment $\langle a \rangle \vdash G \langle a' \rangle$. Assume that the abstract interpretation I is consistent with the verification framework, and that the judgment $\{\gamma(a)\} \vdash G \{\gamma(a')\}$ justifies a copy propagation transformation from G to a program G'. Then, for every*

certified judgment $\{\Phi\} \vdash G \{\Psi\}$ *we have a certified judgment* $\{\gamma(a) \wedge \Phi\} \vdash G' \{\gamma(a') \wedge \Psi\}$.

Consider again the substitution of $\mathsf{Kernel}\langle Z = VectAdd(B[0,k], C[0,k]\rangle$ by the statement $\mathsf{Kernel}\langle Z = VectAdd(A[m, m+k], C[0,k]\rangle$ induced by $\langle a\rangle \vdash s \langle a'\rangle$ such that $B[0,k] = A[m, m+k] \in a$. The interpretation $\gamma(a)$ is such that $\gamma(a) \Rightarrow \gamma(B[0,k] = A[m, m+k])$. Hence, to show that this atomic substitution is justified by the analysis consists on requiring the validity of the following proposition:

$$\gamma(B[0,k] = A[m, m+k]) \wedge \phi[^{B[0,k],C[0,k]}/_{Z}] \Rightarrow \phi[^{A[m,m+k],C[0,k]}/_{Z}]$$

6.5 Sequoia Specific Optimizations

A common characteristic of the optimizations considered in the following sections is that they are defined as a substitution of a subprogram g by another subprogram g' in a bigger program G. We denote with $G[\bullet]$ the fact that G is a program with a *hole*. Given a program g, we denote with $G[g]$ the program obtained by replacing the hole $\bullet$ with g. Then, optimizations are characterized by subprograms g and g', defining a transformation from a program $G[g]$ into a program $G[g']$. The following general lemma complements the following results on certificate translators explained in the rest of this section.

Lemma 6.18. *Let* $G[\bullet]$ *be a program with a hole,* g, g' *programs and* Φ, Ψ *logic formulae. If the judgment* $\{\Phi\} \vdash G[g] \{\Psi\}$ *is certified, then the derivation of the latter contains a certificate for the judgment* $\{\phi\} \vdash g \{\psi\}$*, for some* ϕ *and* ψ*. If there is a certificate for the judgment* $\{\phi\} \vdash g' \{\psi\}$*, then we can construct a certificate for the judgment* $\{\Phi\} \vdash G[g'] \{\Psi\}$.

6.5.1 SPMD Distribution

Consider a program that executes multiple times a single piece of code represented by a subprogram g. If every execution of g involves and independent portion of data, the tasks can be performed in any sequential order or in parallel. SPMD distribution is a common parallelization technique that exploits this condition distributing the tasks among the available processing units.

Programs of the form $\mathsf{Forall}\ j = 0 : k.n - 1\ \mathsf{do}\ g$ are candidates for SPMD distribution, since $k.n$ instances of the single subprogram g are executed in parallel along the range of the iteration variable j. Furthermore, for each value of the iteration value j, the subprogram g operates over an independent partition of the data, as assumed for every program subject to verification.

G' is transformed from G by applying SPMD distribution if G' is the result of substituting every subprogram $\mathsf{Exec}_i(\mathsf{Forall}\ j = 0 : k.n - 1\ \mathsf{do}\ g)$ by the equivalent subprogram $\mathsf{Group}(G_1 \parallel \ldots \parallel G_k)$, with G_i defined as the program $\mathsf{Exec}_i(\mathsf{Forall}\ j = i.n : (i+1)n - 1\ \mathsf{do}\ g)$ for all $i \in [0, k-1]$.

Normally, a real compiler will also consider whether it is convenient to span the computation of g over other child nodes. However, since orthogonal to the transformation of the verification judgment, we do not consider this issue.

Lemma 6.18 in combination with the following lemma that states that the local substitutions defining SPMD distribution preserve certified judgments, implies the feasibility of certificate translation.

Lemma 6.19. *Given a certified judgment* $\{\Phi\} \vdash \mathsf{Exec}_i(\mathsf{Forall}\ j = 0 : k.n - 1\ \mathsf{do}\ g)\ \{\Psi\}$ *it is possible to generate a certified judgment* $\{\Phi\} \vdash \mathsf{Group}(G_1 \parallel \ldots \parallel G_k)\ \{\Psi\}$*, where* G_i *is defined as* $\mathsf{Exec}_i(\mathsf{Forall}\ j = i.n : (i+1)n - 1\ \mathsf{do}\ g)$ *for any* $i \in [0, k-1]$.

Proof. To prove this lemma, we show that a certificate of the judgment $\{P\} \vdash G\ \{Q\}$ corresponding to a subprogram of the form $G = \mathsf{Exec}_i(\mathsf{Forall}\ j = 0 : k.n - 1\ \mathsf{do}\ g)$ can be transformed into a certificate of the judgment $\{P\} \vdash G'\ \{Q\}$, where $G' = \mathsf{Group}(G_1 \parallel \ldots \parallel G_k)$, with $G_i = \mathsf{Exec}_i(\mathsf{Forall}\ j = i.n : (i+1)n - 1\ \mathsf{do}\ G)$ for each $i \in [0, k-1]$. For simplicity, we refrain from considering the application of the subsumption rule **[CSS]**. Then, by application of rule **[E]**, Q is equal to $\downarrow^i(Q')$ for some Q', and the judgment $\{\uparrow^i(P)\} \vdash \mathsf{Forall}\ j = 0 : k.n - 1\ \mathsf{do}\ G\ \{Q'\}$ is certified. We know then that, by application of the rule **[F]**, $Q' = \sum_{j'=0}^{k.n-1} \mathsf{weak}_j(Q_{j'})$, that $\uparrow^i(P) = \bigwedge_{j'=0}^{k.n-1} \mathsf{wp}_{j:=j'}(P_{j'})$ and that for each $j' \in [0, k.n-1]$ we have a certified judgment $\{P_{j'}\} \vdash G\ \{Q_{j'}\}$. For each $i \in [0, k-1]$ we construct a derivation for the judgment $\{\uparrow^i(P)\} \vdash \mathsf{Forall}\ j = i.n : (i+1)n - 1\ \mathsf{do}\ G\ \{Q'_i\}$ where $Q'_i = \sum_{j'=i.n}^{(i+1)n-1} \mathsf{weak}_j(Q_{j'})$, by application of the rule **[F]** and **[CSS]** with a certificate of $\uparrow^i(P) \Rightarrow \bigwedge_{j'=i.n}^{(i+1)n-1} \mathsf{wp}_{j:=j'}(P_{j'})$. By application of rule **[E]** we have a derivation for $\{P\} \vdash \mathsf{Exec}_i(\mathsf{Forall}\ j = i.n : (i+1)n-1\ \mathsf{do}\ G)\ \{\downarrow^i(Q'_i)\}$. Finally, by a simple application of the rule **[G]** we get the certified judgment $\{P\} \vdash \mathsf{Group}(G_1 \parallel \ldots \parallel G_k)\ \left\{\sum_{i=1}^{k}(\downarrow^i(Q'_i))\right\}$. To complete the proof notice that by definition of $\sum$ and $\downarrow^i$, $\sum_{i=0}^{k-1}(\downarrow^i(\sum_{j'=i.n}^{(i+1)n-1} \mathsf{weak}_j(Q_{j'})))$ implies $\downarrow^i(\sum_{j'=0}^{k.n-1} \mathsf{weak}_j(Q_{j'}))$.

Example:

Consider again the program G_{Add} of Section 6.2.4. Assume that at the level of the memory hierarchy at which G_{Add} is executed there are k available child processing units, and that $n = k.m$ for some m. Then, we are interested in distributing the independent computations along the iteration range $[0, n-1]$ splitting them in k subsets of independent computations in ranges of length m. We obtain then, after applying SPMD distribution to program G_{Add}, the following transformed program:

$$
\begin{aligned}
G'_{\mathrm{Add}} := \ & \mathsf{Exec}_0(\mathsf{Forall}\ i = 0 : m-1\ \mathsf{do}\ \mathrm{Add}) \\
& \parallel \mathsf{Exec}_1(\mathsf{Forall}\ i = m : 2m-1\ \mathsf{do}\ \mathrm{Add}) \\
& \quad \cdots \\
& \parallel \mathsf{Exec}_{k-1}(\mathsf{Forall}\ i = (k-1)m : k.m-1\ \mathsf{do}\ \mathrm{Add})
\end{aligned}
$$

Applying the result stated above, we can transform the derivation of the judgment $\{\mathsf{true}\} \vdash G_{\mathrm{Add}}\ \{\mathrm{Post}\}$ into a derivation of $\{\mathsf{true}\} \vdash G'_{\mathrm{Add}}\ \{\mathrm{Post}\}$, proving that the verification judgment is preserved. Recall that we can derive the judgment

$$
\{\mathsf{true}\} \vdash \mathsf{Exec}_r(\mathsf{Forall}\ i = r.m : (r+1)m-1\ \mathsf{do}\ \mathrm{Add})\ \left\{ \uparrow^r \Big(\sum_{r.m \le j < (r+1)m} Q_j \Big) \right\}
$$

for every $0 \le r < k$. One more application of rule **[G]** allows us to derive the judgment $\{\mathsf{true}\} \vdash G'_{\mathrm{Add}}\ \left\{ \sum_{0 \le r < k} \uparrow^r \left(\sum_{r.m \le j < (r+1)m} Q_j \right) \right\}$. Finally, requiring a certificate of the distributivity of $\uparrow^r$ over the operator $+$, and a certificate for $\sum_{0 \le r < k} \sum_{r.m \le j < (r+1)m} Q_j \Rightarrow \sum_{0 \le j < k.m} Q_j$ we get by rule **[SS]**

$$
\{\mathsf{true}\} \vdash G'_{\mathrm{Add}}\ \left\{ \left(\sum_{0 \le j < k.m} \uparrow^r Q_j \right) \right\}\ .
$$

By the reasoning in Section 6.2.4 we have $\sum_{0 \le j < k.m} \uparrow^r Q_j \Rightarrow \bigwedge_{0 \le j < n} \uparrow^r Q_j$, and finally by subsumption rule we get $\{\mathsf{true}\} \vdash G'_{\mathrm{Add}}\ \{\mathrm{Post}\}$.

6.5.2 Exec Grouping

An Exec operation pushes the execution of a piece of code down to one of the memory subtrees. Since the cost of transferring code and data between different levels of the hierarchy is not negligible, there is an unnecessary overhead when several Exec operations contain code with short execution time. Hence, there is a motivation to reduce the cost of invoking code in child nodes, by grouping the computation defined inside a set of Exec operations into a single Exec operation.

We say that a program G' is the result of applying Exec grouping, if it is the result of replacing a set of Exec operations targeting the same child node, by an single and semantically equivalent Exec operation. More precisely, every subprogram $\mathsf{Group}(\{\mathsf{Exec}_i(G_1), \ldots, \mathsf{Exec}_i(G_k)\} \cup H)$ such that $(\mathsf{Exec}_i(G_j))_{j=1}^{k}$ are maximal in the dependence graph and mutually independent, is substituted by the equivalent subprogram $\mathsf{Group}(\{\mathsf{Exec}_i(\mathsf{Group}(\{G_1, \ldots, G_k\}))\} \cup H)$. Then, the dependence relation that defines the graph $\{\mathsf{Exec}_i(\mathsf{Group}(\{G_1, \ldots, G_k\}))\} \cup H$ must be accordingly updated. More precisely, if the subprogram $g \in H$ originally depends on G_i for some $i \in [1, k]$ then g depends on the subprogram $\mathsf{Exec}_i(\mathsf{Group}(\{G_1, \ldots, G_k\}))$ in the modified dependence graph.

The following result expresses that a certified judgment corresponding to set of independent Exec operations can be translated to a certified judgment

for the result of merging the Exec operations into a single one. This result, together with Lemma 6.18, implies the existence of a certificate translator for Exec grouping.

Lemma 6.20. *Consider a set of mutually independent tasks $G_1, \ldots, G_k$ and a dependence graph $\{\mathsf{Exec}_i(G_1), \ldots, \mathsf{Exec}_i(G_k)\} \cup H$ s.t. $(\mathsf{Exec}_i(G_j))_{1 \leq j \leq k}$ are maximal elements. Assume that $\{\Phi\} \vdash \mathsf{Group}(\{\mathsf{Exec}_i(G_1), \ldots, \mathsf{Exec}_i(G_k)\} \cup H)\ \{\Psi\}$ is a certified judgment. Then, it is possible to generate a certified judgment $\{\Phi\} \vdash \mathsf{Group}(\{\mathsf{Exec}_i(\mathsf{Group}(\{G_1, \ldots, G_k\}))\} \cup H)\ \{\Psi\}$.*

Proof. To prove this lemma we show that for a subprogram G of the form

$$\mathsf{Group}(\{\mathsf{Exec}_i(G_1), \ldots, \mathsf{Exec}_i(G_k)\} \cup H)$$

with G_j maximal elements, and certified judgment $\{P\} \vdash G\ \{Q\}$, the judgment $\{P\} \vdash G'\ \{Q\}$ is also certified, with G' equal to

$$\mathsf{Group}(\{\mathsf{Exec}_i(\mathsf{Group}(\{G_1, \ldots, G_k\}))\} \cup H)\ .$$

For simplicity, we do not consider the application of the subsumption rule **[CSS]**, and we assume the existence of the certificate $\mathsf{distrib}_{+,\oplus_i} :\vdash \phi \oplus_i (\psi + \varphi) \Rightarrow (\phi \oplus_i \psi) + (\phi \oplus_i \varphi)$, built by application of the operators of the proof algebra, and by definition of $\oplus_i$. If we consider the last application of rule **[G]**, we know there are sets X and H' such that $X \cup \{\mathsf{Exec}_i(G_j) \mid j \in [1,k]\}$ are the maximal elements in H, $H' = H \setminus (X \cup \{G_i\})$, and we have the certified judgments $\{P\} \vdash G_j\ \{Q_j\}$ for $j \in [1,k]$, $\{P\} \vdash G\ \{Q_g\}$ for $g \in X$, and $\left\{\sum_{j\in[1,k]} Q_j + \sum_{g\in X} Q_g\right\} \vdash \mathsf{Group}(H')\ \{Q\}$. It can be seen that if $X \cup \{\mathsf{Exec}_i(G_j) \mid j \in [1,k]\}$ is the set of maximal elements in H, then $X \cup \{\mathsf{Exec}_i(\mathsf{Group}(\{G_j) \mid j \in [1,k]\}))\}$ are also the maximal elements in $H' \cup X \cup \{\mathsf{Exec}_i(\mathsf{Group}(\{G_j) \mid j \in [1,k]\}))\}$. Hence, if we show that the judgment $\{P\} \vdash \mathsf{Exec}_i(\mathsf{Group}(\{G_j) \mid j \in [1,k]\}))\ \left\{\sum_{j\in[1,k]} Q_j\right\}$ is certified, by definition of $\sum$, and application of the rule **[G]** we get the desired result. To show that $\{P\} \vdash \mathsf{Exec}_i(\mathsf{Group}(\{G_j) \mid j \in [1,k]\}))\ \left\{\sum_{j\in[1,k]} Q_j\right\}$ is a certified judgment, we analyze the derivation of $\{P\} \vdash \mathsf{Exec}_i(G_j)\ \{Q_j\}$ for each $j \in [1,k]$. We have then that $Q_j = P \oplus_i Q'_j$ for some Q'_j, and the certified judgment $\left\{\uparrow^i(P)\right\} \vdash G_j\ \left\{Q'_j\right\}$. Since G_j are independent subprograms, by application of the rule **[G]** we get the certified judgment $\left\{\uparrow^i(P)\right\} \vdash \mathsf{Group}(\{G_j \mid j \in [1,k]\})\ \left\{\sum_{j\in[1,k]} Q'_j\right\}$, and by application of **[E]**, the certified judgment $\{P\} \vdash \mathsf{Exec}_i(\mathsf{Group}(\{G_j \mid j \in [1,k]\}))\ \left\{P \oplus_i \sum_{j\in[1,k]} Q'_j\right\}$. Finally, from $\mathsf{distrib}_{+,\oplus_i}$ we have a certificate for $P \oplus_i \sum_{j\in[1,k]} Q'_j \Rightarrow \sum_{j\in[1,k]} (P \oplus_i Q'_j)$, which together with an application of rule **[CSS]** enables us to certify the judgment $\{P\} \vdash \mathsf{Exec}_i(\mathsf{Group}(\{G_j \mid j \in [1,k]\}))\ \left\{\sum_{j\in[1,k]} Q_j\right\}$.

6.5.3 Copy Grouping

Commonly, for the execution environments targeted by Sequoia programs, transferring several fragments of data to a different level of the memory hierarchy in a single copy operation is more efficient that transferring each fragment of data in a separate operation. For this reason, and since array copy operations are frequent in programs targeting data intensive applications, it is of interest to cluster a set of copy operations involving small and independent regions of the memory into a single transfer operation.

Naturally, this transformation may require an analysis to detect whether two copy operations referring to regions of the same array are indeed independent. However, for simplicity, we consider the case in which the original set of small copy operations are performed over different array variables.

Consider a subprogram $g = \mathsf{Group}(H \cup \{g_1, g_2\})$, where $g_1 = \mathsf{Copy}(A_1, B_1)$ and $g_2 = \mathsf{Copy}(A_2, B_2)$ are mutually independent and maximal in H. Copy propagation consists on substituting g by the equivalent program g' defined as $\mathsf{Group}(H \cup \{g_{1,2}\})$, where $g_{1,2}$ is a copy operation that merges atomic programs g_1 and g_2 into a single transfer operation. In addition, the dependence relation on $\mathsf{Group}(H \cup \{g_{1,2}\})$ must be updated accordingly, such that $g'' \in H$ depends on $g_{1,2}$ iff g'' depended on g_1 or g_2 in the original dependence graph.

Lemma 6.21. *Consider the programs g and g' as defined above. Then, from a certified judgment $\{\Phi\} \vdash g \{\Psi\}$ we can construct a certified derivation for the judgment $\{\Phi\} \vdash g' \{\Psi\}$.*

Proof. To prove this lemma we show that for every subprogram G of the form $\mathsf{Group}(H \cup \{g_1, g_2\})$ transformed into the program $G' = \mathsf{Group}(H \cup \{g_{1,2}\})$, we can certify the judgment $\{P\} \vdash G' \{Q\}$ from a certificate of the judgment $\{P\} \vdash G \{Q\}$. For simplicity, we refrain ourselves from considering applications of the subsumption rule [**CSS**]. If we do not consider [**CSS**], the last rule applied for the derivation of $\{P\} \vdash G \{Q\}$ is [**G**]. Then, for some sets X and H', $X \cup \{g_1, g_2\}$ is the set of maximal elements in $H \cup \{g_1, g_2\}$, and $H' = H \backslash X$. In addition, we have the judgments $\{P\} \vdash g_1 \{Q_1\}$, $\{P\} \vdash g_2 \{Q_2\}$, and $\left\{Q_1 + Q_2 + \sum_{g \in X} Q_g\right\} \vdash \mathsf{Group}(H') \{Q\}$ and a judgment $\{P\} \vdash g \{Q_g\}$ for every g in X. It can be seen that the dependence conditions do not change when merging the copy operations g_1 and g_2. Hence, $X \cup \{g_{1,2}\}$ is the set of maximal elements in $H \cup \{g_{1,2}\}$. Based on this conditions, and by application of rule [**G**] and by associativity of $+$, it sufficient to show that we can certify the judgment $\{P\} \vdash T_{g_{1,2}} \{Q_1 + Q_2\}$. To this end, since we have certificates for $P \Rightarrow T_{g_1}(Q_1)$ and $P \Rightarrow T_{g_2}(Q_2)$ (the only applicable rules are [**A**] and [**CSS**]) and the certificate c, we can construct a certificate for $P \Rightarrow T_{g_{1,2}}(Q_1 + Q_2)$ to certify the judgment $\{P\} \vdash g_{1,2} \{Q_1 + Q_2\}$.

The program G' is the result of applying copy grouping to program G, if every subprogram of the form g in G is replaced by g', where g and g' are as characterized above. The existence of certificate translators follows from Lemma 6.18.

Example:

Consider again the program G_{Add} of Section 6.2.4, that adds two arrays. From the definition of the subprogram Add, we can see that it is a candidate for a copy grouping transformation, since it can be replaced by the equivalent subprogram Add$'$ defined as $\mathsf{Group}(\mathrm{CopyAXBY}; \mathrm{AddP}; \mathrm{CopyZC})$, where CopyAXBY is defined as $\mathsf{Copy}^{\downarrow}(A[i.S, (i+1)S], B[i.S, (i+1)S], X[i.S, (i+1)S], Y[i.S, (i+1)S])$. Assume that judgment of for the example in Section 6.2.4 is certified. To translate this result after applying the transformation above, we must certify the judgment $\{\mathsf{true}\} \vdash \mathrm{CopyAXBY}\ \{Q_{AX} + Q_{BY}\}$. To this end, we reuse the certified judgments $\{\mathsf{true}\} \vdash \mathrm{CopyAX}\ \{Q_{AX}\}$ and $\{\mathsf{true}\} \vdash \mathrm{CopyBY}\ \{Q_{BY}\}$ that are included in the certificate for the judgment $\{\mathsf{true}\} \vdash G_{\mathrm{Add}}\ \{\mathrm{Post}\}$, where Q_{AX} and Q_{BY} are respectively defined as

$$\big(\forall k,\ 0 \leq k < S \Rightarrow X[k] = A^{\uparrow}[k + i.S]\big)$$

and

$$\big(\forall k,\ 0 \leq k < S \Rightarrow Y[k] = B^{\uparrow}[k + i.S]\big)\quad .$$

The fact that makes the translation through, is the validity of the formula $\mathsf{wp}_{\mathrm{CopyAX}}(\phi) \wedge \mathsf{wp}_{\mathrm{CopyBY}}(\psi) \Rightarrow \mathsf{wp}_{\mathrm{CopyAXBY}}(\phi + \psi)$.

Part III

Related Work and Conclusion

Related Work

Certifying Compilation.

Certifying compilation is an extension of a standard compiler that automatically generates a proof that the compiled code satisfies some specific properties. A certifying compiler is built from three main components: A preliminary step consists in propagating basic information, such as the result of type inference, from the source level to the compiled program. This is in some way convenient since a higher level of abstraction is more adequate for the inference of particular properties. However, this can be disadvantageous if the goal is to reason about low-level properties, such as resource consumption of the concrete execution environment.

Another step consists in inferring loop invariants, possible reusing information inferred about the source program if preserved by the compilation. Notice that requiring automaticity on loop invariance inference is a significant obstacle that restricts the complexity of properties that can be considered. Finally a special purpose VCgen is applied to the result of the compilation to generate a set of proof obligations, the proof of which ensure that the executable code adheres to some safety policies. To complete the process, these proof obligations are discharged by an automatic theorem prover.

Automatic generation of certificates comes at a cost. As said before, required properties must be restricted to sufficiently simple safety policies. But in addition, a certifying analyzer may fail to generate a certificate for the output code. However, it does not affect the soundness of the approach.

In addition to generate a certificate ensuring the correctness of the output code, certifying compilers may rely on a static analysis phase to remove unnecessary runtime checks and, thus, improving performance.

The Touchstone compiler [51] is a notable example of certifying compiler, which generates type-safety certificates for a fragment of C. In Chapter 6 of his thesis [52], Necula studies the impact of a particular set of standard program optimizations on certifying compilation, including some of the optimizations considered in this thesis. For each optimization, an informal analysis is made, indicating whether the transformation requires reinforcing the program invariants, or whether the transformation does not change proof obligations. For a particular set of sufficiently simple proof obligations, Necula shows that if the VCgen propagates proof obligations backwards, then the verification conditions are preserved. However, he shows that it is not the case with some more sophisticated transformations like induction variable strength reduction and redundant conditional elimination. The former optimization is a clear example where it becomes necessary to strengthen invariants in order to keep proof obligations provable. Furthermore, Necula shows that in both optimizations the associated program certificate must be modified. While we perform a systematic and implicitly defined transformation of the certificate, Necula relies on the capability of a theorem prover, possibly modulo user-provided

hints, to discharge the modified proof obligation. Even if delegating this task to a powerful theorem prover may be feasible for sufficiently simple safety properties, it is not clear how it scales up to arbitrarily functional properties.

There are many commonalities between his work and ours, but also some notable differences. First, the VCgen used by Necula propagates invariants backwards, whereas ours generates a proof obligation for each invariant. Second, we assume that the program comes with its annotations and certificate, and we have not only to strengthen the annotations, but also to transform the certificate. This is the main difficulty with respect to Necula's work: when he observes that the transformation produces a logically equivalent proof obligation, we have to define a function that maps proofs of the original proof obligation into proofs of the new proof obligation after optimization.

Certified Compilers.

Compiler correctness [32] aims at showing that a compiler preserves the semantics of programs. Traditionally, semantics preservation is understood as the preservation of the input/output behavior of programs; thus most compiler correctness statements are of the form: if running a program p on an initial configuration c returns some final result v, then running the corresponding compiled program $[\![p]\!]$ on the corresponding initial configuration c' shall also return the same final result v.

Because compilers are complex programs, the task of compiler verification can be daunting; in order to tame the complexity of verification and bring stronger guarantees on the validity of compiler correctness proofs, *certified compilation*, see e.g. [17, 44, 21, 65], advocates the use of a proof assistant for machine-checking compiler correctness results. Certified compilation involves modeling formally the source and target languages, their semantics, and providing an executable definition $[\![.]\!]$ of the compiler. Then the correctness statement is expressed in the language of the proof assistant, and proved using the logic of the proof assistant. In proof assistants that support the notion of proof object, the task of turning $[\![.]\!]$ into a certified compiler amounts to build a term H of type:

$$\forall p : \mathsf{Program}, \forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle p, c\rangle \Downarrow v \Longrightarrow \langle [\![p]\!], c\rangle \Downarrow v$$

Certified compilation is not motivated by mobile code, and has not been exploited in the context of PCC architectures. In fact, certified compilation is particularly relevant for the critical software industry, where the code producers and the code consumers belong to the same entity, or trust each other. Nevertheless, it is possible to construct certificate translators from certified compilers, as explained below and suggested independently by Leroy [44].

Under suitable conditions (evaluation is deterministic, compilation preserves nontermination, and programs do not get stuck), one can prove from H that the compiler reflects semantics, i.e., one can build a proof object H' that establishes the dual of preservation of semantics:

$$\forall p : \mathsf{Program}, \forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle [\![p]\!], c\rangle \Downarrow v \Rightarrow \langle p, c\rangle \Downarrow v$$

This proof object may be exploited to transfer evidence from source code programs to compiled programs. Indeed, assume that we want to prove the following property for some program p:

$$\forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle [\![p]\!], c\rangle \Downarrow v \Rightarrow R(c, v)$$

where $R(c, v)$ establishes a formal relation between input configurations and results. Then, if we have constructed the proof object cert_p for

$$\forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle p, c\rangle \Downarrow v \Rightarrow R(c, v)$$

one can build the proof object $\mathsf{cert}_{[\![p]\!]}$ for

$$\begin{aligned} &\lambda c : \mathsf{Config}.\lambda v : \mathsf{Res}.\lambda H_{\text{eval}} : (\langle [\![p]\!], c\rangle \Downarrow v).\ \mathsf{cert}_p(H'\ p\ c\ H_{\text{eval}}) \\ &\quad : \forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ (\langle [\![p]\!], c\rangle \Downarrow v) \Rightarrow R(c, v) \end{aligned}$$

Thus it is in principle possible to build certificate translators from certified compilers. There are however some drawbacks:

— the certificate $\mathsf{cert}_{[\![p]\!]}$ of the compiled program $[\![p]\!]$ encapsulates the definition and correctness proof of the compiler, as well as the source code and its certificate. Hence, the certificate $\mathsf{cert}_{[\![p]\!]}$ is very large, and costly to check. Leroy [44] has suggested that partial evaluation could be used to eliminate much of the proof of correctness of the compiler from the certificate, but the applicability of the method has not been demonstrated;

— traditionally, certified compilers are shown to preserve the input/output behavior of programs, thus the approach rules out properties that are expressed with intermediate assertions or ghost variables. Unfortunately, many interesting properties of programs must be specified using assertions or ghost variables. Thus the approach is restrictive. Leroy [43] has explored means to extend the scope of compiler correctness results beyond input/output behaviors, but his work does not cover yet preservation of intermediate assertions.

Compcert [44, 21] is a notable example of a certified moderately optimizing compiler. The Compcert compiler generates PowerPC code for a significant subset of the C language, and performs several standard optimizations such as constant propagation, common subexpression elimination, register allocation and branch tunneling. The compiler is almost completely specified in the Coq proof assistant, and is accompanied with a proof, in Coq, of a set of lemmas stating that the generated assembly code is semantically equivalent to the input source program. The compiler is executable and can be translated to Ocaml with the extraction mechanism provided by the Coq tool; the performance of the extracted compiler is very reasonable.

Translation Validation.

Translation validation is an approach to verify program transformation by checking, for each individual translation, a correspondence between the output and input programs. It consists mainly of a common model to abstract the semantics of the input and output programs, and an automatic verification procedure that checks whether the semantics of the generated program simulates the semantics of the source program. Since the validation phase is independent of the compiler, i.e., there is no need to specify and verify the compiler, it does not constrain the evolution of the compiler.

Verifying correctness of the compilation consists in giving, for each input program, a proof that the compiled code preserves the semantics of the original one. The difference with verifying the compiler is that instead of establishing once and for all that every output code will be equivalent to the source program, the compiler provides a proof of this equivalence for each compilation result.

Initial work on translation validation [58] proposed a tool to verify the correctness for a small set of proof obligations on a restricted subset of target programs. Necula [53] extended this work by implementing a translation validation infrastructure in the context of a GNU C compiler. It handles the intermediate phases of a realistic compiler, but the set of optimizations under consideration are reduced to structure preserving transformations. Further results on translation validation have presented a more comprehensive set of program optimizations [6, 71]. These include simple structure preserving optimizations as well as non structure-preserving transformations such as loop inversion, loop unrolling, loop fusion and strength reduction. A recent publication [67] presents the development of a formally verified translation validator for instruction scheduling optimizations, specified and verified in the Coq proof assistant.

As discussed in Section III, it is also possible to construct certificate translators from translation validation. Again, assuming we have a notion of proof object, instead of a proof term H such that for any program p:

$$H : \forall p : \mathsf{Program}, \forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle p, c\rangle \Downarrow v \Longrightarrow \langle [\![p]\!], c\rangle \Downarrow v$$

we will have, for every successfully compiled program p, a proof term H_p such that

$$H_p : \forall c : \mathsf{Config}, \forall v : \mathsf{Res},\ \langle p, c\rangle \Downarrow v \Longrightarrow \langle [\![p]\!], c\rangle \Downarrow v$$

The same reasoning explained above with respect to certified compilers permits to build certificate translators from translation validation. This approach has the advantage that a certificate does not encapsulate a definition of the compiler. However, since the output program is not verified independently of the source program, the latter must be delivered as a component of the certificate.

Provable Optimizations through Sound Local Rules.

Rhodium [42] is a special purpose framework aimed at specifying and proving the correctness of program analyses and optimizations. To this end, Rhodium relies on a domain-specific language for specifying the analysis and transformation by means of local rules. A set of statements define the abstract domain of the analysis, called dataflow facts, and specifies its meaning as a predicate on the execution state. Transfer functions of the analysis are specified by the so called propagation rules, that declare the generation or propagation of dataflow facts through single edges. They are specified for a generic instantiation of variables and are local in the sense that they relate only adjacent program nodes. Analyzing the correctness of each local rule consists in verifying it with respect to a single execution step. Rhodium generates, for each local propagation rule, a proof obligation in terms of its semantic interpretation, and then submits them to an automatic prover that attempts to discharge them. Given the validity of every local elementary rule, the validity of the entire analysis follows by a common property of the framework (proved once, by hand, and instantiated for any analysis).

Finally, a set of transformation rules specifies how this inferred data-flow facts are used to trigger program transformations. In the same spirit, giving a formal proof for each elementary transformation rule is sufficient to guarantee the correctness of the whole transformation.

Proof Transforming Compilation.

The Spec# project [5, 41, 40] defines an extension of C# with annotations, and a compiler from annotated programs to annotated .NET files, which can be run using the .NET platform, and checked against their specifications at run-time or verified statically with an automatic prover. The Spec# project implicitly assumes some relation between source and bytecode levels, but does not attempt to formalize this relation. There is no notion of certificate, and thus no need to transform them. Pavlova and Burdy have followed a similar line of work [23] to define a Bytecode Modeling Language (BML), and a VCgen for annotated bytecode programs. Annotations of the Java Modeling Language are translated into BML, and the generated proof obligations are sent to an automatic theorem prover. They present a partial formalization of the relation of verification conditions at source and bytecode levels, but they do not consider proof transformations. Similar results are detailed by Pavlova [57], for a significant subset of Java Bytecode. In a recent work, Barthe, Gregoire and Pavlova [8] establish the preservation of proof obligations from source Java programs to compiled code, obtained by nonoptimizing compilation.

In a similar spirit, Bannwart and Müller [3], provide Hoare-like logics for significant sequential fragments of Java source code and bytecode, and illustrate how derivations of correctness can be mapped from source programs to

bytecode programs obtained by nonoptimizing compilation. Müller and Nordio [48] have developed a proof transforming procedure for a Java to Java Bytecode compiler, in particular, dealing with abrupt termination. They have explained the complications of including a subset of Java with `try-catch`, `try-finally` and `break` statements, and showed how they may be handled. More recently, Nordio, Müeller, and Meyer have formalized [55, 54], using the Isabelle proof assistant, a proof transforming procedure from a subset of the Eiffel programming language to Microsoft's Common Intermediate Language.

Furthermore, there are relevant results on transferring evidence from source programs to compiled programs in scenarios that use alternative technologies for establishing program correctness. These results include type-preserving compilation [13, 66], and Rival's method [59] to translate program invariants generated at source level using abstract interpretation techniques in order to verify safety properties.

Developing further the approach of describing data-flow analysis as type judgments [39], Saabas and Uustalu [62, 61, 60] propose to extend these type-based methods to describe also program transformations. They illustrate the feasibility of the method by explaining in detail two particular transformations: Common subexpression elimination and Dead Variable elimination. They have demonstrated the correctness of both transformations, by derivability of Hoare logic proofs, but also showed a constructive mechanism to transform a Hoare proof of the original program to a Hoare proof of the transformed program.

Another instance of proof preserving compilation is the work of Shao et al. [64]. They define a framework to reason and certify programs beyond simple safety policies but still maintaining properties at a decidable level. In addition, they show that certificates are preserved after applying CPS and closure conversion.

Proof Producing Analysis.

One key component of a certificate translator is the certifying analyzer. In Chapter 2, we have shown, for each standard optimization, that on may define a certifying analyzer straightforwardly, due to the simplicity of the generated proof obligations. It is possible to define a generic certifying analyzer under mild assumptions on the relation between the analysis and the verification infrastructure. This is a substantial improvement if we plan to extend certificate translation to arbitrary semantics preserving transformations.

We are aware of two previous works on certifying, or proof-producing, program analyses. Both consider the backwards case. Seo, Yang and Yi [63] consider a generic backwards abstract interpretation for a simple imperative language and provide an algorithm that automatically constructs safety proofs in Hoare logic from abstract interpretation results. Chaieb [24] considers a flow chart language equipped with a weakest precondition calculus, and provides sufficient conditions of the existence of certificates for solutions of backwards

abstract interpretations. The technique was applied in the context of a certified PCC infrastructure [68].

Conclusion

Certificate translation provides a mechanism to transfer the benefits of source code verification to the certification of mobile code in a PCC scenario. At the cost of interactive verification on the producer side, certificate translation extends the set of properties that can be enforced by PCC to arbitrarily complex functional properties. A compilation procedure is composed of several steps. The first phase consists in a naïve translation from a structured source program to a non-structured lower level intermediate representation. Previous work have proved that a nonoptimizing compiler preserves proof obligations [14, 57]. In this thesis, we have shown that it is not the case in the presence of common optimizations applied along the intermediate compiler steps. To solve this difficulty, we have shown the feasibility of proof transformation in the presence of several program transformations. Furthermore, we have extended our results in certificate translation to less typical verification and program transformation scenarios.

- In chapter 2, we have introduced the principles of certificate translation showing that certificate translators exist for many common optimizations. For concreteness, we focused on a particular set of optimizations applied over a specific intermediate program representation. We have considered optimizations such as constant propagation, common subexpression elimination, loop induction variable strength reduction, register allocation and function inlining. We have provided a classification in terms of the proof transformations needed in order to translate the certificates, and proposed a general certificate translation scheme to cover several optimizations that performs arithmetic simplifications. In addition, we have provided ad-hoc proof transformations for optimizations that fall outside the general scheme. We have introduced the notion of certifying analyzers, an extension of standard analyzers that provide in addition to the static result a certificate of the validity of the analysis, expressed in the underlying verification logic. A certifying analyzer is a necessary component that incorporates the information computed by the static analysis to the original specification, incrementing the original certificate accordingly. We have shown the interaction of certifying analyzers with certificate translation for optimizations as common subexpression elimination and loop induction variable strength reduction.
- In chapter 3, we study certificate translation abstracting away from a particular program representation and a particular verification setting. To that end, we have used a mild extension of abstract interpretation in which a notion of certificates is incorporated to the solutions of a set of

constraints. In this setting, we have isolated a set of sufficient conditions, expressed in an abstract proof algebra, that ensures the existence of certifying analyzers and certificate translators. Our formalization allows us to establish the scalability of certificate translation, since shows that the concept of certificate translation can be potentially instantiated in a wide range of settings, provided the requirements discovered on the abstract model are fulfilled. That is the case in particular with the verification infrastructure of chapter 2.

In addition, we have applied the general framework defined in this chapter to the verification of concurrent programs. As a verification environment, we have presented an Owicki-Gries like logic, in which invariants are shown to be satisfied by an isolated sequential program execution and that it is preserved under the concurrent execution of the other components. In this setting, we have shown that no extra conditions are required to transform certificates in the presence of transformations that operate on sequential components.

- In the second part of the thesis we extend the applicability of certificate translation to less typical scenarios:
  - We have developed a modular verification method for programs with specification-preserving advices, that mildly generalizes the notion of harmless advices of Dantas and Walker, and shown how proof compilation extends naturally to this setting. Our results, while preliminary, establish the feasibility of a Proof Carrying Code scenario with several untrusted intermediaries that enhance the mobile code with specific added value, e.g. related to security and efficiency.
  - Program verification environments increasingly rely on hybrid methods to prove correctness of software. Motivated by applications to Proof Carrying Code, we have shown the coincidence of hybrid verification methods at source and bytecode levels. Additionally, we have shown that hybrid verification methods can be "compiled" into methods based on standard verification condition generation, which ensure that hybrid methods are sound.
  - We have used the framework of abstract interpretation to develop a sound proof system to reason about Sequoia programs, and to provide sufficient conditions for the existence of certificate translators. Then, we have instantiated these results to common optimizations described in [37].

Future Work

There are several directions for future work, in terms of the compiler infrastructure, the complexity of the source and target languages, the power of the static analysis, and its interaction with the verification environment.

Compiler infrastructure: Compilers are commonly defined as a sequence of progressive transformations to the input program. In this thesis, we have

restricted our attention mostly to program optimizations applied in intermediate compiler phases. This is complemented with previous results that have shown that the translation of the source program into an intermediate representation preserves certificates, up to minor differences. Combining this two results is sufficient to assert the existence of certificate translators in a PCC scenario in which applications are distributed as interpretable bytecode, as it is the case for instance of several mobile code environments based on Java Bytecode.

However, there are no complementary results that study proof transformation in the remaining compiler phases, such as register allocation or instruction generation. Further work in this area is necessary to attest the existence of certificate translators for a full compilation procedure. A particular case study that would complement existing work on preservation of proof obligations for Java programs [57] is the case of Java native compilers, that generates native code from a Java bytecode component.

Certifying the analysis: For some optimizations that perform arithmetic simplifications, the analyzer is required to return, in addition to the result of the analysis, a proof attesting its validity in the underlying verification framework. While for some standard optimizations we show that it is relatively simple to generate the required certificates, we have provided a set of sufficient conditions in the abstract domain of the analysis in order to ensure the existence of certifying analyzers. While this set of proof obligations can be easily dischargeable for basic analysis domains such as those employed in common compiler optimizations, it is interesting to study how it extends to more complex settings. For instance, one of such scenarios is the recent method proposed by Halbwachs and Peron [33], to automatically generate invariants referring to array contents.

Independently, it is interesting to consider more sophisticated verification environments in order to dispense analyzers to generate a certificate of their results. The reason that makes essential the representation of the analysis result in the specification logic and a proof of their corresponding verification conditions, is that the VCgen does not consider any information other than the logical specification to generate the proof obligations. Then, the abstract analysis results must be suitably translated in order to be integrated with the original logical specification.

The main motivation to keep as simple as possible the definition of the VCgen is to minimize the computation effort required from the code consumer side.

It would be interesting to estimate how the computational complexity is affected with the development of PCC infrastructures based on hybrid VCgens that take as input the result of an analysis to compute the set of proof obligations. As mentioned in Chapter 5, one of the advantages of such hybrid VCgens is the smaller specification and verification effort. A more important advantage is that the result of the analysis can then be independently checked by a decidable procedure, dispensing analyzers to

certify its results. In consequence, certificate translation does not incur in specification growth, since original invariants are not strengthened with the result of the analysis, and the certificate growth is reduced, since certificates for the analysis are not merged with the original certificate. However, a negative counterpart of this approach is that a complex hybrid VCgen makes the trusted computing base more difficult to trust by the code consumer.

Certificate growth: Another direction for future work is to estimate the certificate growth entailed by a certificate translator. At the level of abstraction in which we have studied certificate translation, we are in a position to estimate the specification growth incurred by a transformation. However, an analysis of the certificate growth is tightly bound to the formal representation of proofs.

Although we have experimented with toy programs and simple optimizations, it would be interesting to perform more extensive experimental results in order to compare, in a realistic scenario, the size of certificates before and after the compiler optimizations. This could be a long an daunting task, because it does not only require the development of a verification environment, a compiler and a certificate translator, but the specification and deductive formal verification of realistic applications.

On the other hand, further research can be pursued in order to reduce the size of certificates after the transformation has been performed. One possibility consists in reducing specifications by means of pruning, as suggested by Besson et al. [20]. The method consists in removing specification fragments that are neither part of the property that the program is required to satisfy, nor auxiliary invariants needed to prove the program correct. Since a transformation in the program specification implies a transformation in the verification conditions, a certificate translator must be defined in the presence of invariant pruning.

Other techniques to control the size explosion consist in reducing the actual representation of a certificate. In the case of a lambda-term representation of certificates, term normalization can be proposed as a technique to generate a smaller proof attesting the same logical formula. It will be an essential contribution to estimate how much it can be gained with this technique in this particular formal representation of proofs.

In this thesis, we aim at providing general schemes to cover certificate translation for an extensive list of compiler optimizations, without abstracting away from implementation issues such as specification or certificate growth. However, the development of alternative proof transformations mechanisms can help attest the applicability of certificate translation to realistic scenarios. Preliminary work has already provided alternative, ad-hoc, proof transformation procedures, to minimize the effect of certificate translation in certificate growth. A prototype implementation has been developed, integrating a tool that translates certificates from a simple imperative language to an RTL representation. This prototype tool has

served to show that certificate growth is negligible for simple optimizations like constant propagation and common subexpression elimination. Further research in this area extending these techniques to more complex optimizations will provide a substantial efficiency improvement in the development of a PCC environment based on certificate translation.

Tool development: Certainly, the implementation of a realistic compiler requires a significant effort, and that is also the case, in a minor extent, of proof assistants and verifications tools. In addition, deductive verification for realistic programs can be a daunting task. Then, a reasonable development project aiming at experimentally estimating the applicability of certificate translation must advocate for the integration of an already existing compiler and verification tools.

Extending compiler analyses with a certification phase and program optimizations with a proof transforming procedure is quite involved. It would then be desirable to count on isolating the definition of the compiler from the certificate translation process. To that end, we can consider an optimization framework that relies on a special purpose specification language for the analysis and optimizations, as has been proposed by the Rhodium project [42]. In this framework, optimization phases are separated from the compiler and, hence, they can be progressively defined and extending the compiler becomes much easier. But more importantly, the set of requirements to define the corresponding certificate translator can be extracted as a set of logical proof obligations and discharged in an independent process.

References

1. E. Albert, G. Puebla, and M. V. Hermenegildo. Abstraction-carrying code. In F. and A. Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning*, number 3452 in Lecture Notes in Computer Science, pages 380–397. Springer-Verlag, 2005.
2. Alpern, B., Carter, L., and Ferrante, J. Modeling parallel computers as memory hierarchies. In *Proc. Programming Models for Massively Parallel Computers*, 1993.
3. F. Y. Bannwart and P. Müller. A program logic for bytecode. *Electronic Notes in Theoretical Computer Science*, 141:255–273, 2005.
4. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.
5. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 151–171. Springer-Verlag, 2005.
6. C. W. Barrett, Y. Fang, B. Goldberg, Y. Hu, A. Pnueli, and L. D. Zuck. Tvoc: A translation validator for optimizing compilers. In K. Etessami and S. K. Rajamani, editors, *CAV*, number 3576 in Lecture Notes in Computer Science, pages 291–295. Springer-Verlag, 2005.
7. G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate translation for optimizing compilers. In K. Yi, editor, *Static Analysis Symposium*, number 4134 in Lecture Notes in Computer Science, pages 301–317. Springer-Verlag, 2006.
8. G. Barthe, B. Grégoire, and M. Pavlova. Preservation of proof obligations for Java. In *International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science. Springer-Verlag, 2008. To appear.
9. G. Barthe and C. Kunz. Certificate translation for specification-preserving advices. In Curtis Clifton, editor, *FOAL*, pages 9–18. ACM, 2008.
10. G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In *European Symposium on Programming*, number 4960 in Lecture Notes in Computer Science, pages 368–382. Springer-Verlag, 2008.

11. G. Barthe, C. Kunz, D. Pichardie, and J. Samborski-Forlese. Preservation of proof obligations for hybrid verification methods. In *Software Engineering and Formal Methods*. IEEE Press, 2008.
12. G. Barthe, C. Kunz, and J. Sacchini. Certified reasoning in memory hierarchies. In G. Ramalingam, editor, *Asian Programming Languages and Systems Symposium*, Lecture Notes in Computer Science. Springer-Verlag, 2008.
13. G. Barthe, D. Naumann, and T. Rezk. Deriving an information flow checker and certifying compiler for Java. In *Symposium on Security and Privacy*. IEEE Press, 2006.
14. G. Barthe, T.Rezk, and A. Saabas. Proof obligations preserving compilation. In T. Dimitrakos, F. Martinelli, P. Ryan, and S. Schneider, editors, *Workshop on Formal Aspects in Security and Trust*, number 3866 in Lecture Notes in Computer Science, pages 112–126. Springer-Verlag, 2005.
15. Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova, and Antoine Requet. JACK: A tool for validation of security and behaviour of Java applications. In *Formal Methods for Components and Objects: Revised Lectures from the 5th International Symposium FMCO 2006*, number 4709 in Lecture Notes in Computer Science, pages 152–174. Springer-Verlag, 2007.
16. N. Benton. Simple relational correctness proofs for static analyses and program transformations. In N. D. Jones and X. Leroy, editors, *Principles of Programming Languages*, pages 14–25. ACM Press, 2004.
17. Y. Bertot, B. Grégoire, and X. Leroy. A structured approach to proving compiler optimizations based on dataflow analysis. In JC. Filliâtre, C. Paulin-Mohring, and B. Werner, editors, *TYPES*, volume 3839 of *Lecture Notes in Computer Science*, pages 66–81. Springer, 2004.
18. F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 364(3):273–291, 2006. Extended version.
19. F. Besson, T. Jensen, D. Pichardie, and T. Turpin. Result certification for relational program analysis. Research Report 6333, IRISA, September 2007.
20. Frédéric Besson, Thomas P. Jensen, and Tiphaine Turpin. Small witnesses for abstract interpretation-based proofs. In *Programming Languages and Systems: Proceedings of the 16th European Symposium on Programming, ESOP 2007*, number 4421 in Lecture Notes in Computer Science, pages 268–283. Springer-Verlag, 2007.
21. S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a c compiler front-end. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.
22. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J.R. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In *Workshop on Formal Methods for Industrial Critical Systems*, volume 80 of *Electronic Notes in Theoretical Computer Science*, pages 73–89. Elsevier, 2003.
23. L. Burdy and M. Pavlova. Java bytecode specification and verification. In *Symposium on Applied Computing*, pages 1835–1839. ACM Press, 2006.
24. A. Chaieb. Proof-producing program analysis. In K. Barkaoui, A. Cavalcanti, and A. Cerone, editors, *ICTAC*, volume 4281 of *Lecture Notes in Computer Science*, pages 287–301. Springer-Verlag, 2006.

25. P. Chalin, J. R. Kiniry, G. T. Leavens, and Erik Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. In Springer-Verlag, editor, *Formal Methods for Components and Objects*, volume 4111 of *Lecture Notes in Computer Science*, pages 342–363, 2006.
26. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.
27. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Principles of Programming Languages*, pages 269–282, 1979.
28. W. J. Dally, F. Labonte, A. Das, P. Hanrahan, J. Ho Ahn, J. Gummaraju, M. Erez, N. Jayasena, I. Buck, T. J. Knight, and U. J. Kapasi. Merrimac: Supercomputing with streams. In *International Conference on Supercomputing*, page 35. ACM, 2003.
29. D. S. Dantas and D. Walker. Harmless advice. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Principles of Programming Languages*, pages 383–396. ACM Press, 2006.
30. K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *Conference on Supercomputing*, page 83. ACM Press, 2006.
31. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Languages Design and Implementation*, volume 37, pages 234–245, June 2002.
32. J. D. Guttman and M. Wand. Special issue on VLISP. *Lisp and Symbolic Computation*, 8(1/2), March 1995.
33. Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 339–348. ACM, 2008.
34. M. V. Hermenegildo and F. Rossi. Strict and nonstrict independent and-parallelism in logic programs: Correctness, efficiency, and compile-time conditions. *J. Log. Program.*, 22(1):1–45, 1995.
35. M. Houston, J. Young Park, M. Ren, T. Knight, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. A portable runtime interface for multi-level memory hierarchies. In M. L. Scott, editor, *PPOPP*. ACM, 2008.
36. U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. Ho Ahn, P. R. Mattson, and J. D. Owens. Programmable stream processors. *IEEE Computer*, 36(8):54–62, 2003.
37. T. J. Knight, J. Young Park, M. Ren, M. Houston, M. Erez, K. Fatahalian, A. Aiken, W. J. Dally, and P. Hanrahan. Compilation for explicitly managed memory hierarchies. In K. A. Yelick and J. M. Mellor-Crummey, editors, *PPOPP*, pages 226–236. ACM, 2007.
38. Shriram Krishnamurthi, Kathi Fisler, and Michael Greenberg. Verifying aspect advice modularly. In Richard N. Taylor and Matthew B. Dwyer, editors, *SIGSOFT FSE*, pages 137–146. ACM, 2004.
39. P. Laud, T. Uustalu, and V. Vene. Type systems equivalent to data-flow analyses for imperative languages. *Theoretical Computer Science*, 364(3):292–310, 2006.
40. K. Rustan M. Leino. Specifying and verifying programs in spec#. In I. Virbitskaite and A. Voronkov, editors, *Ershov Memorial Conference*, volume 4378 of *Lecture Notes in Computer Science*, page 20. Springer, 2006.

41. K. Rustan M. Leino and W. Schulte. Exception safety for c#. In *SEFM*, pages 218–227. IEEE Computer Society, 2004.
42. Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 364–377, New York, NY, USA, 2005. ACM.
43. X. Leroy. Coinductive big-step operational semantics. In *Programming Languages and Systems: Proceedings of the 15th European Symposium on Programming, ESOP 2006*, volume 3924 of *Lecture Notes in Computer Science*, pages 54–68. Springer-Verlag, 2006.
44. X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. G. Morrisett and S. L. Peyton Jones, editors, *Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
45. Francesco Logozzo and Manuel Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In Laurie Hendren, editor, *CC*, volume 4959 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2008.
46. C. Marché, C. Paulin-Mohring, and X. Urbain. The Krakatoa tool for certification of Java/JavaCard programs annotated with JML annotations. *Journal of Logic and Algebraic Programming*, 58:89–106, 2004.
47. A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, Palaiseau, France, December 2004. `http://www.di.ens.fr/~mine/these/these-color.pdf`.
48. P. Müller and M. Nordio. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 39–46, New York, NY, USA, 2007. ACM.
49. G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.
50. G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In *Operating Systems Design and Implementation*, pages 229–243, Seattle, WA, October 1996. USENIX Assoc.
51. G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Programming Languages Design and Implementation*, volume 33, pages 333–344, New York, NY, USA, 1998. ACM Press.
52. G.C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, October 1998. Available as Technical Report CMU-CS-98-154.
53. George C. Necula. Translation validation for an optimizing compiler. *ACM SIGPLAN Notices*, 35(5):83–94, 2000.
54. M. Nordio, P. Müller, and B. Meyer. Formalizing proof-transforming compilation of eiffel programs. Technical Report 587, ETH Zurich, 2008.
55. M. Nordio, P. Müller, and B. Meyer. Proof-transforming compilation of eiffel programs. In R. Paige, editor, *TOOLS-EUROPE*, Lecture Notes in Business and Information Processing. Springer-Verlag, 2008.
56. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica Journal*, 6:319–340, 1975.
57. M. Pavlova. *Java bytecode verification and its applications*. Thése de doctorat, spécialité informatique, Université Nice Sophia Antipolis, France, January 2007.

58. A. Pnueli, E. Singerman, and M. Siegel. Translation validation. In B. Steffen, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer-Verlag, 1998.
59. X. Rival. Symbolic Transfer Functions-based Approaches to Certified Compilation. In *Principles of Programming Languages*, pages 1–13. ACM Press, 2004.
60. A. Saabas and T. Uustalu. A compositional natural semantics and Hoare logic for low-level languages. *Theoretical Computer Science*, 373(3):273–302, 2007.
61. A. Saabas and T. Uustalu. Type systems for optimizing stack-based code. In M. Huisman and F. Spoto, editors, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 190(1) of *Electronic Notes in Theoretical Computer Science*, pages 103–119. Elsevier, 2007.
62. A. Saabas and T. Uustalu. Program and proof optimizations with type systems. *Journal of Logic and Algebraic Programming*, 77(1–2):131–154, 2008.
63. S. Seo, H. Yang, and K. Yi. Automatic Construction of Hoare Proofs from Abstract Interpretation Results. In A. Ohori, editor, *Asian Programming Languages and Systems Symposium*, volume 2895 of *Lecture Notes in Computer Science*, pages 230–245. Springer-Verlag, 2003.
64. Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Trans. Program. Lang. Syst.*, 27(1):1–45, 2005.
65. M. Strecker. Formal Verification of a Java Compiler in Isabelle. In A. Voronkov, editor, *Conference on Automated Deduction*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77, London, UK, 2002. Springer-Verlag.
66. D. Tarditi, J. Gregory Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A type-directed optimizing compiler for ML. In *Programming Languages Design and Implementation*, pages 181–192. Association of Computing Machinery, 1996.
67. J. Tristan and X. Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. *SIGPLAN Not.*, 43(1):17–27, 2008.
68. M. Wildmoser, A. Chaieb, and T. Nipkow. Bytecode analysis for proof carrying code. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation*, volume 141 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2005.
69. M. Wildmoser, T. Nipkow, G. Klein, and S. Nanz. Prototyping proof carrying code. In J-J. Levy, E. W. Mayr, and J. C. Mitchell, editors, *Theoretical Computer Science*, pages 333–347. Kluwer Academic Publishing, August 2004.
70. H. Xi and S. Xia. Towards array bound check elimination in java tm virtual machine language. In *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 14. IBM Press, 1999.
71. L. D. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. Voc: A translation validator for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 65(2), 2002.